Welcome to DHSI 2019!

Thanks for joining the DHSI community!

In this booklet, you will find essential course materials prefaced by some useful information about getting settled initially at UVic, finding your way around, getting logged in to our network (after you've registered the day before our courses begin), and so on.

Given our community's focus on things computational, it will be a surprise to no one that we might expect additional information online for some of the classes - your instructors will let you know - or that the most current version of all DHSI-related information may be found on our website at dhsi.org.

Do check in there first if you need anything that's not in this coursepak.

To access the DHSI wifi network, simply go into your wireless settings and connect to the “DHSI” network and enter the password “dhsi2019”.

And please don’t hesitate to be in touch with us at institut@uvic.ca or via Twitter at @AlyssaA_DHSI or @DHInstitute if we can be of any help ....
DHSI Wi-Fi

Network name: DHSI
Passkey: dhsi2019
The 2019 schedule is just taking shape nicely! A very few things to confirm, add, etc, still but this is the place to be to find out what is happening when / where …

Sunday, 2 June 2019 [DHSI Registration + Suggested Outings]

If you’re here a day or two before we begin, or staying a day or two afterwards, here are a few ideas of things you might consider doing …

Suggested Outing 1, Botanical Beach (self-organised; car needed)

A self-guided visit to the wet, wild west coast tidal shelf (and historically-significant former research site) at Botanical Beach; we recommend departing early (around 8.00 am) to catch low tide for a better view of the wonderful undersea life! Consider bringing a packed lunch to nibble-on while looking at the crashing waves when there, and then have an afternoon drink enjoying the view from the deck of the Port Renfrew Hotel.

Suggested Outing 2, Butchart Gardens (self-organised)

A shorter journey to the resplendently beautiful Butchart Gardens and, if you like, followed by (ahem) a few minutes at the nearby Church and State Winery, in the Saanich Penninsula. About an hour there by public bus from UVic, or 30 minutes by car.

Suggested Outing 3, SaltSpring Island (self-organised; a full day, car/bus + ferry combo)

Why not take a day to explore and celebrate the funky, laid back, Canadian gulf island lifestyle on SaltSpring Island. Ferry departs regularly from the Schwartz Bay ferry terminal, which is about one hour by bus / 30 minutes by car from UVic. You may decide to stay on forever …

Suggested Outing 4, Paddling Victoria’s Inner Harbour (self-organised)

A shorter time, seeing Victoria’s beautiful city centre from the waterways that initially inspired its foundation. A great choice if the day is sunny and warm. Canoes, kayaks, and paddle boards are readily rented from Ocean River Adventures and conveniently launched from right behind the store. Very chill.

And more!

Self-organised High Tea at the Empress Hotel, scooter rentals, visit to the Royal BC Museum, darts at Christies Carriage House, a hangry breakfast at a local diner, whale watching, kayaking, brew pub sampling (at Spinnaker’s, Swans, Moon Under Water, and beyond!), paddle-boarding, a tour of used bookstores, and more have also been suggested!

9:00 to 4:00

Early Class Meeting: 4. [Foundations] DH For Department Chairs and Deans (David Strong Building C124, Classroom)

Further details are available from instructors in mid May to those registered in the class. Registration materials will be available in the classroom.

3:00 to 5:00

DHSI Registration (MacLaurin Building, Room A100)

After registration, many will wander to Cadboro Bay and the pub at Smuggler’s Cove OR the other direction to Shelbourne Plaza and Maude Hunter’s Pub OR even into the city for a nice meal.

Monday, 3 June 2019

Your hosts for the week are Alyssa Arbuckle, Ray Siemens, and Jannaya Friggstad Jensen.

7:45 to 8:15

Last-minute Registration (MacLaurin Building, Room A100)
### 8:30 to 10:00
- **Welcome, Orientation, and Instructor Overview** *(MacLaurin A144)*
  - Welcome to the Territory
  - Welcome to DHSI: Ray Siemens, Alyssa Arbuckle
  - Welcome from UVic: Jonathan Bengtson (University Librarian), Alexandra D’Arcy (Associate Dean Research, Humanities)

### 10:15 to Noon
<table>
<thead>
<tr>
<th>Class Title</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Foundations] Digitisation Fundamentals and their Application</td>
<td>Clearihue A103, Lab</td>
</tr>
<tr>
<td>[Foundations] Introduction to Computation for Literary Criticism</td>
<td>Clearihue A102, Lab</td>
</tr>
<tr>
<td>[Foundations] Making Choices About Your Data</td>
<td>Digital Scholarship Commons, McPherson Library A308, Classroom</td>
</tr>
<tr>
<td>[Foundations] DH For Department Chairs and Deans</td>
<td>David Strong Building C124, Classroom</td>
</tr>
<tr>
<td>[Foundations] Developing a Digital Project (With Omeka)</td>
<td>Clearihue A031, Lab</td>
</tr>
<tr>
<td>[Foundations] Race, Social Justice, and DH: Applied Theories and Methods</td>
<td>Cornett A229, Classroom</td>
</tr>
<tr>
<td>[Foundations] Fundamentals of Programming/Coding for Human(s)ists</td>
<td>Clearihue A108, Lab</td>
</tr>
<tr>
<td>Out-of-the-Box Text Analysis for the Digital Humanities</td>
<td>Human and Social Development A160, Lab</td>
</tr>
<tr>
<td>Sound and Digital Humanities</td>
<td>Cornett A120, Classroom</td>
</tr>
<tr>
<td>Critical Pedagogy and Digital Praxis in the Humanities</td>
<td>Clearihue D132, Classroom</td>
</tr>
<tr>
<td>Digital Humanities for Japanese Culture: Resources and Methods</td>
<td>McPherson Library A003, Classroom</td>
</tr>
<tr>
<td>Conceptualising and Creating a Digital Edition</td>
<td>McPherson Library 210, Classroom</td>
</tr>
<tr>
<td>Retro Machines &amp; Media</td>
<td>McPherson Library 129, Classroom</td>
</tr>
<tr>
<td>Geographical Information Systems in the Digital Humanities</td>
<td>Clearihue A105, Lab</td>
</tr>
<tr>
<td>Introduction to IIIF: Sharing, Consuming, and Annotating the World’s Images</td>
<td>Cornett A121, Classroom</td>
</tr>
<tr>
<td>Web APIs with Python</td>
<td>Human and Social Development A170, Lab</td>
</tr>
<tr>
<td>Ethical Data Visualization: Taming Treacherous Data</td>
<td>Cornett A128, Classroom</td>
</tr>
<tr>
<td>Linked Open Data and the Semantic Web</td>
<td>Cornett A132, Classroom</td>
</tr>
<tr>
<td>Palpability and Wearable Computing</td>
<td>McPherson Library A025, Classroom</td>
</tr>
<tr>
<td>The Frontend: Modern JavaScript &amp; CSS Development</td>
<td>Clearihue A030, Lab</td>
</tr>
<tr>
<td>Modelling. Virtual. Realities. A Practical Introduction to Virtual (and Augmented) Reality</td>
<td>Human and Social Development A150, Lab</td>
</tr>
<tr>
<td>Information Security for Digital Researchers</td>
<td>David Strong Building C114, Classroom</td>
</tr>
</tbody>
</table>

### 12:15 to 1:15
- **Lunch break / Unconference Coordination Session** *(MacLaurin A144)*
  - (Grab a sandwich and come on down!)

### 1:30 to 4:00
- **Institute Lecture: Jacqueline Wernimont** *(Dartmouth C)*: "Sex and Numbers: Pleasure, Reproduction, and Digital Biopower"
  - Chair: Anne Cong-Huyen *(U Michigan)*
  - *(MacLaurin A144)*

### 4:10 to 5:00
- **Abstract:** Drawing from Numbered Lives *(MIT 2018)*, this talk will consider a long history of sex-number entanglement in Anglo-American Cultures. Drawing on historical and contemporary objects and practices, Wernimont will ask "in what ways do theories of biopower, critical gender and critical race studies, and media studies" suggest that we can understand this set of entanglements and their impacts. NB: While relevant, this talk will not include discussions of sexual trauma or violence. It will include frank discussion of sex acts and various ways of translating sexual behavior into numbers.

### 5:00 to 6:00
- **Opening Reception** *(University Club)*

---

**Tuesday, 4 June 2019**

### 9:00 to Noon
- **Classes in Session**

### 12:15 to 1:15
- **Lunch break / Unconference**
  - "Mystery" Lunches

### 1:30 to 4:00
- **Classes in Session**

**DHSI Conference and Colloquium Lightning Talk Session 1** *(MacLaurin A144)*
Wednesday, 5 June 2019

9:00 to Noon
Classes in Session

Lunch break / Unconference

"Mystery" Lunches

Presentation: An Introduction to Scholarly Publishing with Manifold (MacLaurin A144)
Lunch included for those who [register here]

This presentation introduces Manifold Scholarship, a Mellon-funded digital publishing platform developed by the CUNY Graduate Center, The University of Minnesota Press, and Cast Iron Coding. Manifold allows you to create beautiful, dynamic open access projects that can include text, images, video, embedded resources, and social annotation. We will provide an overview of Manifold and demonstrate how faculty, students and staff in the digital humanities can use Manifold to publish open access scholarly works, conduct and participate in peer review, and create custom edited versions of public domain course texts and OER.

1:30 to 4:00
Classes in Session

4:15 to 5:15
DHSI Conference and Colloquium Lightning Talk Session 2 (MacLaurin A144)
Chair: Kim O'Donnell (Simon Fraser U)

Catherine Ryu (Michigan State U), “Tone Perfect: Developing a Multimodal Audio Database for Mandarin Chinese as an Open Source”


Jessica Linzel (Brock U), “The Shopkeeper Aristocracy: Mapping Trade Networks in Colonial Niagara”

Kirsten Painter (U Washington), “From Bogatyrs to Bread: Digitization & Online Exhibition of Rare Russian Children's Books at the U Washington”

John Barber (Washington State U), “A Mighty Span”

6:00 to 7:00
"Half Way There!" [An Informal, Self-Organized Birds of a Feather Get-Together] (Felicitas, Student Union Building)
Bring your DHSI nametag and enjoy your first tipple on us! [A great opportunity for an interest group meet-up ....]

Thursday, 6 June 2019

9:00 to Noon
Classes in Session

Lunch break / Unconference

"Mystery" Lunches

[Instructor lunch meeting]

1:30 to 4:00
Classes in Session

4:15 to 5:15
DHSI Conference and Colloquium Lightning Talk Session 3 (MacLaurin A144)
Chair: Kim O'Donnell (Simon Fraser U)

Colleen Kolba (U South Florida), “What Comics can Teach our Students about Multimodal Literacy”

Trish Baer (ETCL; U Victoria), “Preserving Digital Legacies: Archived Websites and Digital Discoverability”

Suchismita Dutta (U Miami), “The Importance of Archival Transcription for Genre Building”

Jeffrey Lawler (California State U, Long Beach), “Twining our way through the Past: Video Game Authoring as History Pedagogy”
Sean Smith (California State U, Long Beach), “Gaming the History Curriculum, Games Writing as History Pedagogy in College Classroom”

**Friday, 7 June 2019 [DHSI; ADHO Pedagogy SIG Conference Opening]**

9:00 to Noon  Classes in Session

12:15 to 1:15  Lunch Reception / Course E-Exhibits (MacLaurin A100)

1:30 to 1:50  Remarks, A Week in Review (MacLaurin A144)

2:00 to 3:00  Joint Institute Lecture (DHSI and ADHO Pedagogy SIG Conference):
Matt Gold (CUNY Graduate Center and Association for Computers and the Humanities): “Thinking Through DH: Proposals for Digital Humanities Pedagogy”
Chair: Diane Jakacki (Bucknell U) (MacLaurin A144)

Abstract: How do we teach digital humanities, and how should DH be taught? What, indeed, should we teach when we teach DH? This talk will present a proposal for grounding digital humanities pedagogical practice in the research interests of our students and the epistemological foundations of our methods rather than through an approach grounded more central in data and methods.

3:30 to 5:00  Joint Reception: DHSI and ADHO Pedagogy SIG Conference (University Club)
E-Poetry Event (Chris Tanasescu)
Watch this space for details, including how to participate!

**Saturday, 8 June 2019 [Conference, Colloquium, and Workshop Sessions]**

8:00 to 9:00  Conference / Workshop Registration (MacLaurin A100)

The day’s events are included with your DHSI registration. If you’re not registered in DHSI, you’re very welcome to join us by registering here as a Conference / Colloquium / Workshop participant. We’ll have a nametag waiting for you!

Coffee, Tea, &c?  Looking for some morning coffee or tea, or a small nibble? Options and hours of operation for weekend campus catering are available here. Mystic Market usually opens around 10.00.

9:00 to 4:00  DHSI Conference and Colloquium Sessions
ADHO Pedagogy SIG Conference Sessions
Right2Left Workshop Sessions

9:00 to 4:00  All Day DHSI Workshop Session (click for workshop details and free registration for DHSI participants)
55. Introduction to Machine Learning in the Digital Humanities [8-9 June; All day, each day] (David Strong Building C124, Classroom)

9:00 to 9:10  Informal Greetings, Room Set-up (Lobby, outside Hickman 105)

**Session 1**
DHSI Colloquium and Conference (Hickman 105)
Digital Humanities & Literature, Chair: Kim O’Donnell (Simon Fraser U)
- Youngmin Kim (Dongguk U), “Transdiscursivity in the Convergence of Digital Humanities and World Literature”
- Caroline Winter (U Victoria), “Digitizing Adam Smith’s Literary Library”
- Kaitlyn Fralick (U Victoria); Kailey Fukushima (U Victoria); Sarah Karlson (U Victoria), “Victorian Poetry
9:10 to 10:30

ADHO Pedagogy SIG Conference (Hickman 110)
Chair: Katherine Faull (Bucknell U)
- Aaron Tucker and Nada Savicevic (Ryerson U), "Write Here, Right Now: An Open Source eTextbook for the Flipped Classroom"
- Heather McAlpine (U Fraser Valley), "Digital Meters: Using Text Encoding to Teach Literature in the Undergraduate Classroom"
- Tiina H. Airaksinen (U Helsinki), "Digital Humanities in Cultural Studies: Creating a MOOC course for University Students and A-Level Students"

Right2Left Workshop (Hickman 116)
Keynote - Nathan P. Gibson (Ludwig Maximilians U, München): "Thinking in -JTR: Reorienting the Directional Assumptions of Global Digital Scholarship"

10:30 to 10:40

Break

10:40 to Noon

Session 2

DHIS Colloquium and Conference (Hickman 105)
Digital Humanities & Society, Chair: Eleanor Reed (Hastings C)
- Joel Zapata (Southern Methodist U), "Uncovering the Southern Plains’ Mexican American Civil Rights Movement"
- Ayo Oesianwo (U Ibadan), "Online Newspaper Construction of Agitation for the Sovereign State of Biafra in Nigeria"
- Joseph Jones (U British Columbia), "Testbed for an Approach to Distant Reading: Fictions That Represent Vietnam War Resisters in Canada"
- Brendan Mackie (U California, Berkeley), "Visualizing Long-Term Cultural Change: An Example From The Birth of Civil Society"

ADHO Pedagogy SIG Conference (Hickman 110)
Chair: Laura Estill (St Francis Xavier U)
- Jane Jackson (Chinese U of Hong Kong), "Interrogating digital spaces for intercultural meaning-making"
- Ryan Ikeda (UC Berkeley), "Disrupting Digital Literacy: Situating Electronic Literature Among Public Education Initiatives"
- Christopher Church, Katherine Hepworth (U Nevada, Reno), "We’re STEAMed! A call for balancing technical instruction and disciplinary content in the digital humanities"
- Chelsea Milbourne (Cal Poly, San Luis Obispo), "Finding the Right Fit between Technology and Class Content: Reflections on Including Web Development in a Digital Storytelling Course"

Right2Left Workshop (Hickman 116)
- Edward "Eddie" Surman (Claremont Graduate U), "Qualitative Digital Text Analysis and #Right2Left Languages: A Demonstration of Atlas.ti using the Hebrew Bible"

Noon to 1:10

Lunch (We recommend Mystic Market on weekends!)

Session 3

DHIS Colloquium and Conference (Hickman 105)
Digital Humanities & Community, Chair: Claire Carlin (U Victoria)
- Pia Russel (U Victoria); Emily Stremel (U Victoria), "Mentorship and disability: Supporting disabled employees in digital humanities"
- Amy Lueck (Santa Clara U), "Virtually Emplacing Indigenous Memory"
- Md. Shehabul Alam (National U Bangladesh), "Integrating Library Service with Union Information and Service Center: A Joint Initiative towards Digital Bangladesh"
- Veronica Gomez (Instituto de Humanidades y Ciencias Sociales (HuCSo) - UNL-CONICET), "Latin American E-literature and Location: The Nation Revisited in Electronic Literature Organization (ELO)"

ADHO Pedagogy SIG Conference (Hickman 110)
Chair: Chris Tănăsescu (UC Louvain)
- Laura Estill (St Francis Xavier U), "One Assignment, Three Ways: Assessing DH Projects in a Literature Course"
- Felix Bayode Oke, Stella N. Kpolugbo (Anchor U Lagos), "The Multimodal Technique as a Pedagogical Tool in Pelu Awofeso’s White Lagos: A Definitive and Visual Guide to the Eyo Festival"
- Francesca Giannetti (Rutgers U, New Brunswick), "So near while apart: Correspondence Editions as Critical Library Pedagogy and Digital Humanities Methodology"

Right2Left Workshop (Hickman 116)
- Najla Jarkas (American U Beirut) and David Joseph Wrisley (NYU Abu Dhabi), "RTL Software Localization and Digital Humanities: the Case Study of Translating Voyant Tools into Arabic"
2:30 to 2:40 Break

Session 4

DHSI Colloquium and Conference (Hickman 105)
- Ashleigh Casserence-Stanfield (U Chicago), “Sonifying Hamlet and Reading the Room”

ADHO Pedagogy SIG Conference (Hickman 110)
Chair: Aaron Tucker (Ryerson U)
- Youngmin Kim (Dongguk U), “Teaching Digital Humanities and World Literature in Class”
- Alice Fleerackers, Juan Pablo Alperin, Esteban Morales, Remi Kalir (Simon Fraser U, U Colorado Denver), “Online annotations in the classroom: How, why, and what do students learn from annotating course material?”
- Andie Silva (York C and Graduate Center, CUNY), “Keeping it Local: Undergraduate DH as Feminist Practice”

Right2Left Workshop (Hickman 116)
- Joanna Byszuk (Institute of Polish Language, Polish Academy of Sciences, Warsaw/Computational Stylists Group) and Alexey Khismatulin (Institute of Oriental Manuscripts, Russian Academy of Sciences, Saint Petersburg), “Attribution of Authorship for Medieval Persian Quasidas with Styometry”

2:40 to 4:00

2:40 to 4:00

Sunday, 9 June 2019 [Workshop Sessions]

8:00 to 5:00

The day’s events are included with your DHSI registration. If you’re not registered in DHSI, you’re very welcome to join us by registering here as a Conference / Colloquium / Workshop participant. We’ll have a nametag waiting for you!

Coffee, Tea, &c?

Looking for some morning coffee or tea, or a small nibble? Options and hours of operation for weekend campus catering are available here. Mystic Market usually opens around 10.00.

9:00 to 4:00

All Day Workshop Sessions (click for workshop details and free registration for DHSI participants)

55. Introduction to Machine Learning in the Digital Humanities [8-9 June; All day, each day] (David Strong Building C124, Classroom)
56. Pedagogy of the Digitally Oppressed: Anti-Colonial DH Methods and Praxis [9 June; All Day] (Hickman 116, Classroom)
57. Natural Language Processing and Network Coding Apps for Text & Textual Corpus Analysis in the Humanities [9 June; All Day] (David Strong Building C114, Classroom)

AM Workshop Sessions (click for workshop details and free registration for DHSI participants)

9:00 to Noon

59. 3D Visualization for the Humanities [9 June; AM] (Cornett A229, Classroom)
60. It’s All Relational: AbTeC’s Indigenous Video Game Workshops as Storytelling Praxis [9 June; AM] (Cornett A121, Classroom)
61. Spatial DH: De-Colonizing Cultural Territories Online [9 June; AM] (Clearihue D130, Classroom)
62. DIY Digital Editions: Workflow + Philosophy [9 June; AM] (Clearihue D132, Classroom)
63. Creating a CV for Digital Humanities Makers [9 June; AM] (David Strong Building C108, Classroom)

Noon to 1:00

Lunch (We recommend Mystic Market on weekends!)

1:00 to 4:00

PM Workshop Sessions (click for workshop details and free registration for DHSI participants)

65. Indigenous Futurities in the Classroom and Beyond [9 June; PM] (Cornett A121, Classroom)
66. DHSI Knits: History of Textiles and Technology [9 June; PM] (Fine Arts 109, Classroom)
67. Book History Pedagogy Using Scalar [9 June; PM] (Cornett A229, Classroom)
68. Linked Open Datafication for Humanities Scholars [9 June; PM] (McPherson Library A003, Classroom)
69. Stylo - WYSIWYM Text Editor for Humanities Scholars [9 June; PM] (McPherson Library A025, Classroom)

After the day, many will wander to Cadboro Bay and the pub at Smuggler’s Cove OR the other direction to Shelbourne Plaza and Maude Hunter’s Pub OR even into the city for a bite to eat.

Monday, 10 June 2019
Your hosts for the week are Ray Siemens and Jannaya Friggstad Jensen.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:45 to 8:15</td>
<td>DHSI Last-minute Registration (<a href="#">MacLaurin A100</a>)</td>
</tr>
<tr>
<td>8:30 to 10:00</td>
<td>Welcome, Orientation, and Instructor Overview (<a href="#">MacLaurin A144</a>)</td>
</tr>
<tr>
<td><strong>10:15 to Noon</strong></td>
<td>Classes in Session (click for details and locations)</td>
</tr>
<tr>
<td></td>
<td>- 30. [Foundations] Databases for Digital Humanists (<a href="#">McPherson Library 210, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 33. Digital Storytelling (<a href="#">Cornett A120, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 34. Text Mapping as Modelling (<a href="#">Cleanihue D131, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 35. Stylometry with R: Computer-Assisted Analysis of Literary Texts (<a href="#">Cleanihue A102, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 36. Open Access and Open Social Scholarship (<a href="#">Cleanihue D130, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 37. Digital Games as Tools for Scholarly Research, Communication and Pedagogy (<a href="#">Cornett A229, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 38. Queer Digital Humanities (<a href="#">David Strong Building C114, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 40. Introduction to Electronic Literature in DH: Research and Practice (<a href="#">Cornett A128, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 41. Surveillance and the Critical Digital Humanities (<a href="#">David Strong Building C108, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 42. Text Analysis with Python and the Natural Language Toolkit (<a href="#">Cleanihue A103, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 43. Creating LAMP Infrastructure for Digital Humanities Projects (<a href="#">Human and Social Development A170, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 44. Processing Humanities Multimedia (<a href="#">Human and Social Development A150, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 46. Digital Humanities Pedagogy: Integration in the Curriculum (<a href="#">Cornett A121, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 47. Accessibility &amp; Digital Environments (<a href="#">Priestly Law Library 265, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 48. Agile Project Management (<a href="#">Cornett A132, Classroom/Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 49. XPath for Processing XML and Managing Projects (<a href="#">Cleanihue A105, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 50. Endings: How to End (and Archive) your Digital Project (<a href="#">Priestly Law Library 192, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 51. Text Processing - Techniques &amp; Traditions (<a href="#">McPherson Library A025, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 52. Introduction to Humanities Data Analysis &amp; Visualization in R (<a href="#">HDA, Human and Social Development A160, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 53. Introduction to Network Analysis in the Digital Humanities (<a href="#">Cleanihue D132, Classroom</a>)</td>
</tr>
<tr>
<td><strong>12:15 to 1:15</strong></td>
<td>Lunch break / Unconference Coordination Session (<a href="#">MacLaurin A144</a>) (Grab a sandwich and come on down!)</td>
</tr>
<tr>
<td></td>
<td>&quot;Mystery&quot; Lunches</td>
</tr>
<tr>
<td><strong>1:30 to 4:00</strong></td>
<td>Classes in Session</td>
</tr>
</tbody>
</table>
|              | Institute Lecture: Angel David Nieves (San Diego State U): "3D Mapping and Forensic Traces of Testimony: Documenting Apartheid-Era Crimes Through the Digital Humanities"  
<p>|               | Chair: Constante Crompton (U Ottawa) (<a href="#">MacLaurin A144</a>)             |
|              | Abstract: In 1989 the killing of a queer, 14-year-old youth in Winnie Mandela's house named Stompie Seipei (an event that few in South Africa are willing to recall, let alone discuss, in any detail) -- is perhaps one of the most glaring examples where the queer and activist community was suppressed or erased from anti-apartheid/liberation histories. Digital humanities may actually help both reconstruct and recover a history that is still very early in the telling, despite what is commonly believed about the liberation struggle and the contributions of queer activists in the dismantling of apartheid. Perhaps it could explain why a youth such as Seipei was killed -- or at the very least, provide a more complex and messy narrative that permits one to know more how the history of queer anti-apartheid activists was suppressed. This talk outlines a methodology for &quot;messy thinking and writing&quot; in the digital humanities that -- through a queer and feminist intersectional framework -- permits a more complex layering of oral histories and 3D historical reconstructions. |
| <strong>5:00 to 6:00</strong> | Reception (<a href="#">University Club</a>)                                       |
| <strong>Tuesday, 11 June 2019</strong> |                                                                 |
| <strong>9:00 to Noon</strong> | Classes in Session                                                      |
| <strong>12:15 to 1:15</strong> | Lunch break / Unconference                                              |
|              | &quot;Mystery&quot; Lunches                                                      |</p>
<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:15 to 5:15</td>
<td>DHSI Conference and Colloquium Lightning Talk Session 4</td>
</tr>
<tr>
<td></td>
<td>Chair: Lindsey Seatter (U Victoria)</td>
</tr>
<tr>
<td></td>
<td>• Ashley Caranto Morford (U Toronto); Kush Patel (U Michigan); Arun Jacob (McMaster U), “#OurDHIs anticolonial: Questions and challenges in dismantling colonial influences in digital humanities pedagogy”</td>
</tr>
<tr>
<td></td>
<td>• Julia King (U Bergen), “Developing Network Visualizations of Syon Abbey's Books, 1415-1539”</td>
</tr>
<tr>
<td></td>
<td>• Luis Meneses (ETCL, U Victoria), “Identifying Changes in the Political Environment in Ecuador”</td>
</tr>
<tr>
<td></td>
<td>• Alicia Brown (Texas Christian U), “Digital Cartography of the Ancient World”</td>
</tr>
<tr>
<td></td>
<td>• Laura Horak (Carleton U), “Building the Transgender Media Portal”</td>
</tr>
<tr>
<td></td>
<td>• Andrew Boyles Peterson (Michigan State U), “Last Mile Tracking: Implications of Rental Scooter Surveillance”</td>
</tr>
<tr>
<td>6:00 to 8:00</td>
<td>DHSI Newcomer's Gathering</td>
</tr>
<tr>
<td></td>
<td>Come down, buy meal and a beverage, and make some new friends!</td>
</tr>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference</td>
</tr>
<tr>
<td></td>
<td>&quot;Mystery&quot; Lunches</td>
</tr>
<tr>
<td></td>
<td>Presentation: An Introduction Jupyter Notebooks for Researchers</td>
</tr>
<tr>
<td></td>
<td>This presentation introduces Jupyter Notebooks for researchers, via a partnership between Compute Canada and the Pacific Institute for the Mathematical Sciences (PIMS) including a large number of Canadian institutions. Read more here. Presenting is James Colliander, PIMS Director and team.</td>
</tr>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:15 to 5:15</td>
<td>DHSI Conference and Colloquium Lightning Talk Session 5</td>
</tr>
<tr>
<td></td>
<td>Chair: Lindsey Seatter (U Victoria)</td>
</tr>
<tr>
<td></td>
<td>• Calin Murgu (New College of Florida), “Putting local metadata to strategic use: A Dashboard for visualizing 60 years of theses metadata”</td>
</tr>
<tr>
<td></td>
<td>• Jason Lajoie (U Waterloo), “Queer Critical Making and the Logic of Control”</td>
</tr>
<tr>
<td></td>
<td>• John Barber (Washington State U), “Zambezi River Bridge”</td>
</tr>
<tr>
<td></td>
<td>• Kent Emerson (U Wisconsin-Madison), “Digital Mappa and the George Moses Horton Project”</td>
</tr>
<tr>
<td>6:00 to 7:00</td>
<td>&quot;Half Way There (yet again)!&quot; [An Informal, Self-Organized Birds of a Feather Get-Together]</td>
</tr>
<tr>
<td></td>
<td>Bring your DHSI nametag and enjoy your first tipple on us! [A great opportunity for an interest group meet-up ....]</td>
</tr>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference</td>
</tr>
<tr>
<td></td>
<td>&quot;Mystery&quot; Lunches</td>
</tr>
<tr>
<td></td>
<td>[Instructor lunch meeting]</td>
</tr>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:10 to 5:00</td>
<td>Institute Lecture: Karina van Dalen-Oskam (Huygens Institute and U Amsterdam; Alliance of Digital Humanities Organizations): “The Riddle of Literary Quality: Some Answers”</td>
</tr>
<tr>
<td></td>
<td>Chair: Aaron Mauro (Penn State, Behrend C)</td>
</tr>
</tbody>
</table>
|             | Abstract: What is literature, and can you measure it? That is the key question of the project The Riddle of Literary Quality. "The Riddle" is a research project of the Huygens Institute for the History of the Netherlands (Amsterdam) in collaboration with the Fryeke Akademy (Leeuwarden) and the Institute for Logic, Language and Computation (University of Amsterdam). The Riddle combines computational analysis of writing style with the results of a large online survey of readers, completed by almost 14,000 participants. In my talk, I will go into
some of the main results of the project.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch Reception / Course E-Exhibits (MacLaurin A100)</td>
</tr>
<tr>
<td>1:30 to 2:00</td>
<td>Closing, DHSI in Review (MacLaurin A144)</td>
</tr>
</tbody>
</table>

**Contact info:**
e institut@uvic.ca  P: 250-472-5401  F: 250-472-5681
Agile Project Management in the Digital Humanities
Contents

1 Introduction 5

2 Past Experience 9

3 Minimum Viable Project 13

4 Features, Stories, and Milestones 17

5 Building a Roadmap 19

6 Executing 23

7 Sizing 27

8 Prioritizing 31

9 Being Agile 35
1

Introduction

Agile project management is about negotiating the completion of a project from beginning to end while remaining flexible. Being patient and delaying decisions until you have to make them, gathering as much information as you can in the mean time, and then taking action with the information you have, always keeping alternatives in mind in case your first plan of action doesn’t pan out. Just as a fighter shifts from foot to foot to be ready to counter a punch, the agile project manager constantly considers shifts to accommodate any changes in the project’s environment. But it’s about more than just negotiating within the rules. It’s about changing the rules of the game to better ensure a successful project.

We don’t need to use agile management for everything. If we know exactly what we need to do, and we have the time and money to do it, then we just do it. But if all of the questions aren’t answered, or we don’t have more time and money than we need, or we have a “garden of forking paths” before us, then agile methods can help us stay focused and do those things that are most likely to bring success. Understanding our audience’s needs (chapter 3), creating stepping stones to our goals (chapter 4), scheduling and calendaring (chapter 5), breaking tasks down into smaller tasks and understanding how long they might take (chapter 7), and prioritizing (chapter 8). All of these by themselves are valuable, but when brought together into a single workflow (chapter 6), they become powerful in reshaping even a personal project.

Schedule

I have designed the week so that we can discover how agile project management helps us focus on our project’s success. The schedule in Table 1.1 provides the broad outline. Rather than dive into theory with lectures, we will discuss different aspects of project management drawing on our own experience and needs. Using agile project
methods, we will tweak each day based on what we want to explore. By the end of the week, we should have a solid understanding of what’s at stake in our projects so that we can share with others on our teams.

<table>
<thead>
<tr>
<th>Morning</th>
<th>Afternoon</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Monday</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>What is a project? How are projects managed?</td>
<td>What is a minimum marketable feature (MMF)?</td>
<td>What are the minimum marketable features for your project?</td>
</tr>
<tr>
<td>What’s a pivot?</td>
<td>What are your MMFs?</td>
<td></td>
</tr>
<tr>
<td>What happens when someone publishes something that causes you to reconsider part of your project?</td>
<td>How does the MMF depend on the audience?</td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>What do you need to do to accomplish your MMF?</td>
<td>Who does what, when? How do we build a calendar so that we know when we can expect certain project milestones?</td>
<td>Build a draft roadmap for your project listing all major milestones.</td>
</tr>
<tr>
<td>How do these goals or milestones depend on each other? How long does each one take? Who’s responsible for each one?</td>
<td>Where are we flexible in our schedule?</td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>How do we execute our roadmap?</td>
<td>How much work can get done in a sprint?</td>
<td>Describe how you break your project milestones into sprints, if you do, and how you size work in each sprint.</td>
</tr>
<tr>
<td>What does the day-to-day look like in an agile project?</td>
<td>How do we size tasks? How big is too big? How small is too small?</td>
<td></td>
</tr>
<tr>
<td><strong>Thursday</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>What are our budgets and other constraints?</td>
<td>What happens when things don’t go as planned?</td>
<td>Create a prioritization rubric for your project. This rubric should address how each project audience perceives the impact of a task.</td>
</tr>
<tr>
<td>How do we accomplish our MVP within these constraints? How do we prioritize, and for whom?</td>
<td>How do we compensate?</td>
<td></td>
</tr>
</tbody>
</table>

Friday morning is reserved for lab time. DHSI wraps up Friday afternoon.

<table>
<thead>
<tr>
<th>Table 1.1: Schedule</th>
<th></th>
</tr>
</thead>
</table>

I’ve written a chapter in this workbook for each section in the schedule. I’ve included a few exercises as provocations. We will use these exercises to introduce our projects to each other and explore how we can make them more flexible while ensuring delivery of the critical work. All of your work for these exercises may be posted to the workshop’s community. This private community allows us to share and comment on each other’s work. It also serves as a community in which we can ask questions and provide feedback after the workshop is over.

We should have about an hour or two of discussion each morning or afternoon centered on different topics as shown in the schedule. The rest of the time is for each of you to work through the exercises as prompts to help you create your own agile project plan. If topics come up that are of interest to the entire workshop, we can come together for more discussion.

Finally, I include at the end of each chapter a list of references and further reading. I don’t intend for this collection to be exhaustive. It’s a starting point in our exploration of the world of agile project management. We can bring more into the discussions based on our interests.

**Exercise 1**: Introduce yourself to the workshop by telling us a little about your background and the primary project or area of interest that you would like to focus on for the week.
Following this workbook, I’ve included a number of pages from Wikipedia and elsewhere that are available under a Creative Commons license. These articles are great resources to get more details on particular methods or concepts. Not all of them will apply to your project, so don’t worry about conflicting suggestions or try to do everything at once. After the workshop, you’ll be able to use these resources to find further guidance in specific techniques.

Which Agile?

Agile is many things to many people. Every team implements it a bit differently. But then, every team is unique. No other team has exactly the same people with the same skills as your team. How you socialize and share the work impacts the kind of agile you find most comfortable.

The agile manifesto\(^1\) gives us a foundation to guide us. We will focus on individuals and interactions rather than processes and tools. Working software rather than documentation. Collaboration rather than contract negotiation. And maintaining flexibility to respond to change rather than blindly follow a predetermined plan. This doesn’t mean that those other things aren’t needful, but they aren’t as important in delivering a good project.

The agile methods that we will discuss this week are based on a few assumptions about the kinds of projects we encounter in the humanities. We assume a deadline, a small team, and a fixed budget. Many of the methods we discuss can be ignored if we don’t have a deadline or if our deadline is far into the future, and if we can augment our budget over time. Even without those constraints, we can do agile project management. We just have a lot more flexibility.

I won’t be teaching you all of the theory around agile. Instead, I will share what I have found useful in my experience as a starting point for discussion. Does your team need something different? Then let’s figure out what a good first guess might be.

With agile, as with jazz, there are no wrong answers as long as you and your team can continue delivering on your project.

References and Readings


Past Experience

When learning a new way of doing things, we should reflect on our past. What have we seen work well before? What have we seen fail? What would we have changed? Why? How would we want to do it differently in hindsight?

These are more than just questions to bridge between our past experience and the workshop. They are at the heart of agile project management. At the end of each work sprint, we take a few moments to hold a retrospective with the team. This is a time to listen to the team. Similar to receiving feedback in a writing workshop, the proper response from a manager is, “Thank you.” If we are managing the project, then we are managers. Maybe the team needs more detail in the tasks, or maybe someone’s vacation had an impact that we hadn’t foreseen. We constantly tweak how we work so that we become better at what we do as a team. Between the end of one work sprint, and its retrospective, and the beginning of the next work sprint, we take the feedback that we receive, reflect on it, and adjust the rules for the next work sprint. So we need to start our week by asking these questions about our past experience so that we can find where we need to adjust the rules of the game.

Common Problems

McConnell categorizes problems as people-related, process-related, product-related, or technology-related. Here are some common problems that projects run into. We’ll see how agile project management can guard against these. What are some other problems you’ve seen?

Paralysis by analysis. A project is paralyzed by analysis when work can’t proceed until all of the questions have been answered. Analysis is a process of accessing and mitigating risk. Some analysis is necessary, but if the team can’t proceed until the analysis...
is completed, then that’s a whole team not working on the project. Too much analysis too early in the project is a hallmark of waterfall project management. Rather than find and plan for all of the pitfalls the team might run into, it’s better to be agile (chapter 9): encourage flexibility in the team so that when a problem does arise, the team is confident in its ability to solve the problem.

The “death march” project. A project becomes a “death march” when it continues going with no end in sight. Each day is doing more work regardless of progress. The project’s not done until it’s done. Every last item. Everything is a priority and needs to get done eventually. We avoid this by knowing our ultimate goal (chapter 3) and creating a roadmap (chapter 5).

The team experiences burn out. A project threatens to burn out team members with overtime when everything has to get done by a particular deadline. Everything is a priority, and nothing can be dropped. The only way to achieve this is to work overtime week after week. We avoid this by knowing how much we can do in a day, week, etc. (chapters 7 and 8).

The team thrashes from one goal to another. A project thrashes when its goals shift from day to day. For example, user research might indicate that a feature that’s in progress needs to be designed differently. It could be due to shifting priorities from someone with power outside of the project. Any number of reasons can cause priorities to shift. We can avoid thrashing by defining what we are doing in a work sprint and making sure that any new tasks are noted but delayed until a future work sprint (chapter 6).

A few years ago, I was part of a team that was developing a set of features for an important partnering customer. We had gone through several rounds of development and feedback, typical of user-centered agile development, and were now at the point of sending some of our user experience researchers to the customer to watch them using our product. Each evening, around 4:30, we would have a conference call to discuss the results of the on-site observations. At the end of the meeting, we would realize that we needed a new feature in our software to improve the user experience, or else risk losing the customer. So a few of us would hammer out the changes and push them to production by the next morning, ready for a new round of observations. Sometimes, we put a change in, tested it, and realized we shouldn’t have made the change.

Exercise 2: Find a project on the web that you find interesting and that has some impact on your project. For example, it could be on the same general topic as your project, could use some techniques you’re thinking about using, or could use some of the same data you’re using. This project needs to publish information on how it has been managed so that you can see how the management of the project might impact its results. Write a short post summarizing the project and its management style. Include a link to the project. Read the posts by others in the workshop and comment.

Exercise 3: Describe how this project makes you rethink something about your own project.

We weren’t taking the time to analyze the observations and ask what impact the changes might have. We simply observed and executed. Without solid principles to guide our decisions, we ended up thrashing about. Working long hours with no well-defined goal lead to burn out and the feeling of being on a “death march.”

Grants and Funding

Often, granting agencies want to know what they are getting in return for providing funding. They want to make sure you can deliver, so they ask for a breakdown in terms of milestones or intermediate goals, the people involved and what they bring to the project, and how much money it will take to deliver. These kinds of questions encourage bad project planning by pushing you to have all of the answers before you begin.

But there is hope. By focusing on the big picture—the goals—and perhaps some milestones to demonstrate that the goals are not trivial, you can deliver the big picture while forgoing some of the polish or non-critical aspects of the project.

References and Readings


When we start a new project, we want to identify the smallest and simplest outcome that might be useful to us. We can build on and complicate this with more features as time and funding allow, but we can’t reduce this core and still justify any of the work in the project. This core is our minimum meaningful feature set. In the parlance of startups, this is the minimum marketable feature: the simplest and smallest set of features that has any hope of being useful to a customer.

You will also hear people talk about a minimum viable product or project (MVP). The MVP is the smallest and simplest outcome that is useful to you, but not necessarily meaningful to your target audience. An MVP is great for getting a feel for how complicated a project might be when you’re working on a grant proposal. It might even serve as a proof of concept to show people what you envision.

Every project has its own way of defining its MMF. The easiest way is to sketch out all of the possible features you want in your project and then judge each one based on its priority. If the project lacks the feature, can it still be worth spending time and money on the project? If so, then the feature is not part of the MMF. Don’t discard the feature, but set it aside for now and recognize that you can add it back if time allows. Otherwise, it’s fodder for a follow-on project. Ultimately, the MMF is the core set of features, results, data, etc., without which the entire effort would be considered a waste of time by anyone outside of the project.

What if a project needs a feature because we promised it to a funding agency in the grant application, but no one else would miss it? Then we can either negotiate with the funding agency to remove that feature from the requirements, which might work in this case, but not for others, or recognize that different groups will need different features.

**Exercise 4:** List all of the features in your project. What kind of result does the feature support (e.g., exploration, data, narrative)?
Audience

When judging each feature, do so with the intended audience or audiences in mind. Different audiences expect different features. Some audiences, such as the project team, do not consume the MMF but do consume the MVP. We distinguish between these audiences and the ones we listed above by specifying external audiences and internal audiences.

Internal Audiences

Your internal audience is made up of all of the people working day to day to make your project a success. You might have all of these on a larger project. Or, if you’re working alone, you may do some of all of these roles at various times. Your internal audience is interested in the MVP because it allows your audience to estimate complexity, resource requirements, and other aspects of building the MMF.

- **software engineer** — the software engineer (SWE) builds any software that the project relies on to deliver its features.
- **user experience designer** — the user experience (UX) designer works with the external audiences of the project to make sure the project addresses the needs of those audience members in a way that works for them.
- **site reliability engineer** — the site reliability engineer (SRE) works with the internal audiences of the project to make sure the project addresses the needs of those audience members in a way that works for them and increases the reliability of the project for external audiences.
- **product owner** — the product (or project) owner (PO) understands the overall design of the project and knows how each of the smaller goals tie into the overall project. The PO manages the priorities of the team to ensure that the most valuable work gets done.

External Audiences

Your external audience is made up of all of the people outside of your team who want your project to be a success. You might have all of these, or only a few, but your external audience doesn’t depend on the size of your team. Your external audience is interested in the MMF because it gives them something valuable that they won’t have without your effort.

- What are some specific external audiences?
• The subject matter expert — the subject matter expert (SME) is someone outside of the project who is an expert in the project topic. The SME does not need as many explanations of terms and relationships in project data. The SME may provide critical feedback.

• The colleague — the colleague is someone down the hall who isn’t part of the project but competes for funding or other resources. The colleague might be a SME, but is more likely to be focused on other topics in the same field.

• The public — the public user is not a SME in the project topic but has an interest in the outcome. The public does not approve or deny any financial support for the project. Rather, the public only consumes the results without any expectation to provide critical feedback.

• The funder — the funder is the granting agency or other material supporter without whom the project could not proceed. The funder approves the project goals when giving financial support and often expects those goals to be satisfied.

Prioritizing Audiences

We have to prioritize our audiences. Some audiences are more important to us than others. This can be due to funding—we want to apply for and receive funding from the granting agency again—or due to the primary purpose of our project. Whatever the reason is, being aware of it helps us understand how to prioritize different work (see chapter 8 [page 31] for more information on prioritizing).

What outcomes or features does each audience need in order to consider the project not a waste of time? Note that this is different than considering the project a success or exceeding expectations. The first hurdle is simply not to waste time. Once we know that we aren’t going to waste time, then we can consider what else we need to do to build on that work to make the project a success. This also lets us know where we have to hold the line or else when other factors come into play. For example, if funding gets cut, this lets you know if you must abandon any further effort or pivot to some other goal that can be reached with the remaining budget.

What outcomes or features does each audience need in order to consider the project a success? This is the second hurdle after ensuring the project isn’t a waste of time. It’s additional work that adds value to the absolute minimum core.

Finally, what outcomes or features would have each audience consider the project a wild success beyond their expectations? These

Exercise 5: What are your external audiences? Which audience matters the most? Which matters the least?

Exercise 6: Which audience is most important for each of the features in your project?
are stretch goals that can be added if time and resources allow, but shouldn’t get in the way of delivering other priorities.

Exercise 7: Write a short description about your project and its minimum viable product. Is the MVP the data gathered by the project, its presentation, or a set of papers interpreting the results?

Readings and References


Now that we know our MVP, MMF, and success goals, how do we get there? Large projects can be overwhelming. How do we break the project down into manageable pieces so we don’t get overwhelmed?

Let’s say that we wanted to build a freeway across the continent. We would need to decide on a route and acquire the land. We would need to clear the land, grade everything to the right level, lay down the bed, and finally seal the road to protect it from the elements.

Now if we did this like a lot of programmers start programming, we’d just strike out and start buying land, clearing it, etc., without thinking about where we want to go. Once we got a lot cleared, we’d start building the road. When we got to the other side, we’d realize we needed more land, so we’d try to buy something nearby. Whatever was easiest for us to get, we’d use. We’d have infinite flexibility, but no idea about our ability to get anywhere. For all our effort, we might end up building a circular loop rather than a road across the continent.

On the other hand, we could work like many enterprise, waterfall IT projects. We could be diligent and plan everything down to the last detail. Map out an optimum route and make every effort to buy just the land we needed regardless of the price. We could use the best maps in the world, but even with all of the money at our disposal, we still wouldn’t know if we could make it to the other side of the continent.

In both cases, we are prone to failure because we make decisions at the wrong time and we work at the wrong level of granularity. The former example of a programmer jumping into building something without thinking through any of what might be required is making decisions too late and doesn’t pay attention to any details. The latter example of a waterfall project planning everything out ahead of time is making decisions too soon and at too granular a level.

In an agile project, you want to make decisions, but at the right time and with the right level of granularity. Make them too detailed
and too soon and you don’t have as much information as if you had delayed the decisions. Make them too general and too late, and you’ve wasted time and opportunity while introducing errors by having to make wrong assumptions.

What are the features, stories, milestones, or other pieces of the project that we can use to make the overall project more manageable? We want to avoid thinking about tasks at this stage in our project planning. Rather, focus on the results you want. Features describe the results you want. Milestones describe the way points on the way to creating the features. Stories describe the narratives that tie together the features and milestones.

We should plan first for the MVP and then layer on the MMF. This lets us build something for ourselves that validates the ideas driving the project before we spend more time building something for our external audience. Make sure we can create a sequence of milestones that will result in delivering the MVP and the MMF. Then, we can add more milestones to reach the success goal. These milestones are guideposts that allow us to show results even before we reach something publishable or otherwise distributable outside the research group.

Often, an agile development team will work towards a demo. Each milestone should result in something that can be demonstrated to a close group of associates. What do you expect to demo for each of the milestones?

**Readings and References**

Basecamp. [https://basecamp.com/](https://basecamp.com/).


Pivotal tracker. [https://www.pivotaltracker.com/](https://www.pivotaltracker.com/).

Trello. [https://trello.com/](https://trello.com/).


---

**Exercise 8:** Write short descriptions for some of the milestones or intermediate goals in your project. Are these milestones themselves mini-MMFs?

**Exercise 9:** Rework your project milestones into mini-MMFs. Which of these depend on you alone, and which depend on others? What roles do you need on a team to accomplish these milestones?
5

Building a Roadmap

At this point in your project, you should have a list of features or stories describing what you need to do to reach your goal. You should have a good idea of any interdependencies and areas for further research. Don’t let the need for more research delay the start of your project. Factor the research into your project and plan to make changes as you make discoveries.

Now, we need to see if you can get everything done in the time you have, or with the money in your budget, or some other constraint that could require dropping a feature or facet of your project. We don’t want to go into more detail than this because we want to wait until we have to provide more detail. As we hit each milestone, we will learn more about our project. This will inform how we flesh out the details of subsequent milestones.

A rule of thumb that I use when building a roadmap calendar is to ask each group responsible for a feature or milestone to estimate how much time they need in order to accomplish that milestone or feature. I then double it. This tends to work remarkably well.

If we find that our roadmap runs over the constraints on the project, such as time on a grant, then we can select features or milestones to whittle down, either by shortening them a bit or by eliminating them entirely. On the other hand, if we find that the roadmap leaves time left over at the end, then we can expand some milestones or save the extra time for things we might discover later. We can also declare some features to be “stretch goals” that we can accomplish if we finish all of our higher priority features before we run out of time, money, or other constraining resource.

In some ways, this is the simplest exercise in a project, but it gives us a scaffold for everything else we do after this.

Readings and References

Aha! https://www.aha.io/.
## BrailleSC Calendar

The following schedule assumes 25% effort for ... and 25% effort for ... (100% = 40 hours/week).

### Github Setup (2 weeks)
Set up github repository supporting BrailleSC work, especially with Wordpress Plugin and related sub-projects.

### Library Requirements (3 weeks)
Research LibLouie and PHP to see how the two might be able to work together to produce English-to-Braille translations in a web environment, preferably using UTF-8 instead of alternate encodings.

### Local LibLouie (4 weeks)
Create a PHP library that allows WordPress to use LibLouie without calling out to a remote service. The library should have a sufficiently flexible API to provide access to the affordances surfaced in the previous research milestone. This library will be used in subsequent milestones by Amanda as she develops the Wordpress plugin proper. The PHP library should be a standalone package that can be installed on a system and used outside of Wordpress.

### Wordpress Plugin Requirements (1 week)

<table>
<thead>
<tr>
<th>Activity</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wordpress Plugin Presentation</td>
<td>4 weeks</td>
</tr>
<tr>
<td>Remote Library Requirements</td>
<td>2 weeks</td>
</tr>
<tr>
<td>Wordpress Plugin Configuration</td>
<td>3 weeks</td>
</tr>
<tr>
<td>Remote LibLouie</td>
<td>4 weeks</td>
</tr>
<tr>
<td>Local/Remote LibLouie and Wordpress</td>
<td>5 weeks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Activity</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plugin Publication</td>
<td>5 weeks</td>
</tr>
</tbody>
</table>

---

**Figure 5.1:** Example Project Calendar

Executing

Once we have our roadmap, we’re ready to begin our first sprint. Just as we learned in early composition courses, we need to tell them what we’re going to tell them, tell them, and then tell them what we told them. We begin a sprint by describing the work we plan to do, we continue the sprint by doing that work, and then we end the sprint by looking back and seeing what we did. Different agile methods will have different names for these three parts of the sprint, but they all accomplish the same thing.

Sprint Planning

We bring the team together and introduce the first feature or milestone on which we want to work. We make sure everyone on the team has a common understanding of what we want to deliver. Without this common understanding, a lot of time will be spent communicating one-on-one and playing a game of telephone.

Once everyone is on the same page, then we need to write down all of the tasks that need to be done to accomplish the milestone or feature. These need to be fairly granular. Things that can be done in a day or two. Any bigger than that and we have a difficult time understanding the true complexity of the task. Smaller tasks are easier to work with as well. If a large task is broken up into several smaller tasks, we know better where we are in the process of accomplishing the larger task. If we need to, we can move some of the sub-tasks to a different sprint.

Each team member should participate in creating tasks. It’s not our job as project managers to tell everyone what to do. It’s our job to facilitate. We can guide and we can prompt, but we should create tasks or act as a secretary for the team.

At this point in the process, we should have a good idea of what we are trying to do, how much time we have to do it, and how we will do it. We’ll see in the next chapters how to estimate the size of

A sprint is a span of time in which we focus on a feature or set of features with the intent to have something to demonstrate at the end of the time.
tasks and how to prioritize them for a sprint so we have a better idea of how realistic our schedule is. For now, ordering the tasks based on dependencies is good enough.

For the purposes of this workshop, we assume that we are working on a single feature at a time. This simplifies our work, but doesn’t mean we can’t have multiple features in a single sprint. Instead of having feature planning and sprint planning at the same time, we can plan each feature ahead of time with a smaller team that will be focused on that feature and then bring everyone together with all of the tasks for the features in a single, interwoven list.

**Exercise 11:** As preparation for the first sprint in your project, write a sprint planning document. This document shows the goals of the sprint, who is participating (at least the role if you don’t have someone in mind), and the start and end dates of the sprint.

**In the Sprint**

Once we know what we want to accomplish for the sprint, we accomplish it. That is, we go through the tasks that we have listed in the sprint until we run out of tasks or we get to the end of the sprint. If we run out of tasks while in the sprint, we can pull tasks from the backlog. If we run out of tasks in the backlog, then we need to groom tasks from the icebox.

Sprints can be managed in a number of ways, with Scrum and Kanban being two of the main ways on the spectrum of choices. We can draw on extreme programming for some software development practices, but Scrum and Kanban aren’t as specific to the kind of work we’re doing.

Regardless of the method, agile teams usually do some form of status meeting on a regular basis. This is the standup, a roughly fifteen minute meeting in which all members of the team give a simple status report: what they worked on the day before, what they’re working on today, and any blockers keeping them from making progress. Nothing else. Generally, only those doing work on tasks in the sprint speak. This keeps the meeting short and focused. Anything else that needs discussing should be taken to a meeting scheduled for another time and only involving the people necessary.

We expect to discover new things during a sprint. If that discovery is a new task that doesn’t block any work in the sprint, then we should put that task in the backlog. If it’s otherwise ready to be done, or the icebox, if it needs further research or definition.

**Scrum**

Scrum is a more formal sprint management process from which we get sprint planning, daily standups, and retrospectives.
**Kanban**

Kanban is a sprint management style borrowed from Japanese industry. It’s more fluid than Scrum. Tasks are taken from the top of the sprint queue based on various factors, such as fit of a task to the person looking for a task. Rather than have sprint planning, the team may have feature planning as new features are needed to organize and prioritize tasks, but this is driven by how many tasks are ready to be worked on.

A Kanban system may have several queues: a queue of features ready to be planned, a queue of tasks ready to be worked on, etc. As each queue becomes shallow, work can be pulled from the previous queue. When the list of ready tasks gets low, a new feature can be pulled off of the feature queue and developed into tasks. These tasks are pushed onto the task queue and worked on.

Having a regular retrospective and review is still helpful, but is not as critical as with Scrum if we are attuned to the team. The important thing is to make sure the rules of the game are helping the team do its best work rather than getting in the way. If they are in the way, change them without waiting for a formal meeting.

**Sprint Retrospective**

The goal for any team is to be the best that they can be. We use the retrospective to see how the team is working: what is working well and what could be improved. This is an opportunity for the team to discuss possible changes and make any tweaks that help the team work together better. It is important that the manager listen to the team and not impose their own values or judgements on the team during the retrospective.

**Readings and References**

Sizing

At the end of the day, when we have that last hour left and we want to pick a task that we think we can complete before heading home, how do we know which ones should take less than the time we have? How do we make the best use of our time without having to interrupt ourselves in the middle of something? By estimating the size of our tasks.

Sizing is contentious, and the agile community falls into two camps: those who size, and those who don’t. Sizing can be useful in forecasting and providing a gut check on how realistic a sprint or schedule might be. But at the end of the day, you and your team get done what you get done regardless of how large or small each task is. If you don’t find value in sizing your tasks, then don’t worry about sizing. If you do want to size your tasks, then read on!

We assign a size to each task so that we know how much we can expect to get done in a day or in a sprint. It doesn’t matter if we’ve never done this before, or if our estimates are wildly inaccurate. As we work through several sprints, we’ll come to recognize how long similar tasks will take and our estimates will become more consistent.

We can size tasks in several ways. We can estimate how much time a task will take, or we can estimate how complex it is. The important thing is that the team agree on how and which tasks are sized, both the dimension considered and the process, and what sizes are available.

Some teams play “size poker” with a special deck of cards containing the Fibonacci sequence (1, 2, 3, 5, 8, 13, etc.). Others assign “t-shirt sizes” (small, medium, large, etc.). Some mix different units, assigning t-shirt sizes to features, and Fibonacci numbers to tasks.

Some assign sizes to tasks that represent progress in the project’s features and no size to bugs, since bugs don’t create new features. This encourages a team to get it right the first time so that bugs don’t decrease their productivity.
T-Shirt Sizing

The easiest way to size tasks is using adjectives such as “small,” “medium,” “large,” and “extra large.” Choose adjectives that have an obvious ordering. Decide how big is too big, and break those tasks down into smaller tasks.

Time

If you need to lay out all of the tasks on a timeline, then you might be tempted to size by time. That is, estimate how much time each task should take, and then see if you have enough time to get everything done. This is the type of sizing you encounter in waterfall project planning. You see everything lined up in a GANTT chart indicating critical paths and opportunities for parallel effort.

Unfortunately, these estimates are rough, back of the envelope guesses at how long something will take to complete. Software development isn’t like building a house, where standard procedures are used from pouring the foundation to laying the last shingles on the roof. Especially in a research project, software development is exploratory.

So these estimates should be on an exponential scale similar to t-shirt sizes: 30 minutes, an hour, two hours, four hours, etc. These are rough guides to how long you think it might take to accomplish the task, but don’t be surprised if it takes a lot longer, or a lot less time. Sometimes, the most complex problem can be solved with a few lines of code because the data structures are aligned with the problem. Other times, a simple feature can require massive refactoring of an application.

Fibonacci Sequence

Using the fibonacci sequence is similar to using the time estimates, but without being in terms of time. This makes the sequence useful for measuring tasks by complexity or some other non-temporal dimension. Rather than each subsequent size doubling, it follows a

Readings and References


Exercise 12: Describe how you size things in your project. Do you have different scales for features and tasks? Why?
Prioritizing

If we have all the time in the world, and all the computational forces of the universe arrayed against a problem, then we might be able to afford the right answer regardless of the question. But we don’t. Instead, we have limited time (deadlines if not lifetimes) and limited money (grants or other funding) with which to approach a question. We have to accept “good enough” if “perfect” is out of reach. Fortunately, “good enough” and “perfect” are spots on a spectrum of results. This chapter is about how to select tasks so that we can move from “good enough” towards “perfect.”

How bad is “good enough?” It can’t be worse than our MVP. Remember that our MVP was the minimum viable project. Doing less than that isn’t worth it. So we have to do everything that delivers our MVP. Any task that would keep us from achieving our MVP if it weren’t done is a task that we must do. We have to make these tasks higher priority than anything else we do for our project.

Once we understand what we have to do as part of our MVP, then we can see what else we can add to move ourselves along the spectrum towards “perfect.” For this, we need budgets.

Budgets are about prioritization. If something doesn’t make the budget, it’s because something more important did. The things in a budget are more important than things not in the budget. How do we decide priorities for our tasks? Who are the audiences for our tasks? How does the importance of a task change with the audience?

Prioritization reveals risks. If something doesn’t get done, it’s because something more important got done. If we aren’t willing to drop something, then we haven’t prioritized. If we don’t have too much to do within the budgets we have, then we haven’t pushed boundaries enough, and we risk leaving something important undone that we could have done.
Binary Insertion Prioritization

In the Binary Insertion method, we take each task and consider where it falls in the list of already prioritized tasks. When we find a place where all of the tasks above it are more important, and the ones below it are less important, then we have found its place in the prioritized list. We repeat this for each task until we have all of our tasks prioritized.

Over thinking the tasks can be a danger. The more nuance you consider, the more time you’ll need to prioritize. In the end, having a rough order is better than finding the perfect order.

MoSCoW Prioritization

In the MoSCoW method, we label tasks as Must, Should, Could, or Won’t (or Would). This lends itself to natural prioritization. If we find ourselves using one of these key words when describing our tasks, then we have already assigned a priority.

A task is a must if not doing it will result in our work being a waste of time. A task is a should if not doing it requires a painful workaround, but does not lead to our work being a waste of time. A task is a could if not doing it requires a simple workaround. A task is a won’t if not doing it won’t be noticed.

For example, if the core of our research can be represented by a data set, then the construction of that set might be a must, but the visualization of that data set in a nice web page might be a should because someone can take the data set and visualize it with any number of tools. On the other hand, if the argument you are making depends on a specific visualization, then that visualization might be a must. Or if your target audience can’t install new software, then they don’t have a workaround, painful or not. The visualization becomes a must.

We prioritize for each sprint and start with the most important tasks: the musts. Once those are completed, we move on to the shoulds. Finally, the coulds. At the end of the sprint, we review any remaining tasks and decide either to move them to a new sprint, in which case we go through the prioritization process again in case something has changed, or we choose not to do them at all. If something doesn’t get done, it’s because something more important did get done.

This prioritization process is not perfect. A number of simple workarounds, considered together, could become a painful workaround and be worth fixing as a whole. Each workaround introduces friction into the audience’s experience. Too much fric-
tion will turn them away and make them look elsewhere for similar features (e.g., data or analysis).

**Weighted Shortest Job First**

Rather than focus only on the importance of a particular task or feature, we can add a weight representing the value of completing the task sooner rather than later. Or a weight that represents the cost of delaying completion of the task later rather than sooner. Both are equivalent, but one might be easier to consider than the other.

**Audience**

We’ve already discussed our audience in terms of our minimum viable project (page 14). There, we listed our external audiences, but not our internal ones, namely, the members of our team. A typical software development team has software developers, testers, and operations. In better teams, members shift between these roles, engendering empathy and reducing finger pointing or siloing. In a digital humanities project, you will likely have developers, testers, and operations, though small teams tend to have many more cross-functional members. Commercial software projects have team members focused on the user experience, interviewing users and testing new designs so they can create the best product. We’re more likely to have subject matter experts in our projects.

Now, we need to consider how our audience impacts our priorities.

We should consider the primary audience of our project. For a commercial product, this would be the customer who is paying good money in order to use the product. For a research project, is our primary audience the public? Is it our colleagues? Who do we want to see using the result of our project? Our primary audience is our north star. Ultimately, everything we do has to move our project forward for them. We can use the exercise on page 15 to see how we ranked our external audiences.

**Priority Rubric**

A priority rubric provides rules of thumb for assigning priorities to tasks or breaking ties between tasks based on the tasks’ impact on each audience. The priority of the task should be the highest priority for any audience. For example, if a task is a *must* for testers and a *should* for developers, then the task should be considered a *must* since

---

**Exercise 13**: List all of the audiences for your project. Be sure to capture the external audiences as well as the audiences within your project. Which success criteria depends on which audience?

**Exercise 14**: Create a priority rubric for your project. Describe the impact of not doing a task of a given priority for each audience. Number the audience/priority combinations in order from most important (or critical) to least.
at least one audience will be unable to do their part for the project (e.g., use the software) if the task isn’t done.

<table>
<thead>
<tr>
<th>Audience</th>
<th>Must</th>
<th>Should</th>
<th>Could</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>can’t complete their action.</td>
<td>has a painful workaround to complete their action.</td>
<td>has a simple workaround to complete their action.</td>
</tr>
<tr>
<td>Developer</td>
<td>can’t continue writing or supporting the software.</td>
<td>has a painful workaround to complete their work.</td>
<td>has a simple workaround to complete their work.</td>
</tr>
<tr>
<td>Tester</td>
<td>can’t test a feature.</td>
<td>has a painful workaround to test a feature.</td>
<td>has a simple workaround to test a feature.</td>
</tr>
<tr>
<td>Operations</td>
<td>can’t deploy or publish.</td>
<td>has a painful workaround to deploy or publish.</td>
<td>has a simple workaround to deploy or publish.</td>
</tr>
</tbody>
</table>

Table 8.1: Example MoSCoW Priority Rubric for Typical Software Development (“If the task isn’t done, then the audience . . . ”)

<table>
<thead>
<tr>
<th>Audience</th>
<th>Must</th>
<th>Should</th>
<th>Could</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Developer</td>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Tester</td>
<td>3</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Operations</td>
<td>4</td>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 8.2: Example MoSCoW Priority Order

Readings and References


Being Agile

Agile is about being flexible enough to take on anything that comes our way, be it a budget reduction, unexpected illness, or a new colleague able to contribute. In many ways, being agile in this sense requires a view of the project that is typical of a product owner: someone who has the entire product (or project) in mind and is able to pinpoint how different aspects relate to each other. For example, when someone comes to them with a new feature, they know how that feature might impact other work, or how that new feature might fit into the overall plan.

So we have our MMF, we’ve mapped our milestones, and we’ve started prioritizing the tasks that we can identify as we get started. What happens when things go sideways? When someone gets sick, or a budget gets cut? What gets dropped? What doesn’t? Why? How do we communicate this out to our internal and external audiences?

Agile is about being flexible enough to take on anything that comes our way. We need to be ready to juggle dependencies, move work around, perhaps even milestones, to accommodate unforeseen events. We need to be able to trim features if we are facing a budget or team reduction. We should have new features in our back pocket ready to add if we unexpectedly get more money or people.

Because we don’t do all of our detailed planning up front, we don’t have the sunk costs that would make us fall for the sunk cost fallacy and not want to change course if we had planned a more traditional, waterfall project plan.

Progressive Enhancement

One way to maintain flexibility if we use the MoSCoW prioritization scheme is to break the work for a feature or goal into separate buckets of work based on the priority. This gives us a way to focus on the most needful things first while not forgetting the less important, but still important, work to polish the project.

Exercise 15: You’ve just received word that your budget has been cut by 25%. Which features or tasks need to be dropped? Why? Do you still have a viable project?

Exercise 16: In addition to the 25% budget reduction, you’re now sharing your developer’s time (33%) with a colleague who has agreed to come on as an advisor for your project. They have a related project and need the developer’s time. How does this development impact your project? Do you still have a viable project?
For each goal, we have a bucket of features that keep us from realizing the goal if we don’t build them. We also have a bucket of features that, if not done, have workarounds even if they are painful to use. Finally, we have a bucket of features that, if not done, represent a minor inconvenience. Of course, we also have a bucket of features we won’t build. Likewise, we partition the tasks for a feature into buckets of tasks that we must do, tasks that we should do, and tasks that we could do if we had time.

Once we have a small number of goals and features broken down into these buckets, we can begin work. We’ll discover new things along the way, but we can put them in the right bucket and keep going.

In our first sprint, we can select the Musts and a few Shoulds from a feature that must be done. We always want to put some work in the sprint that we can drop. If we have to get everything done, then everything is a priority. If everything is a priority, then nothing is a priority. Or, if there’s nothing we can’t do, then we haven’t prioritized.

For our second sprint, we can bring in the Shoulds and a few Coulds from our first feature along with the Musts and a few Shoulds from a second feature. We focus on the Musts first because we don’t want to drop one of them if we can drop a Should or a Could.

For our third sprint, we bring in any remaining tasks from our first feature, the Shoulds and some Coulds from our second feature, and the Musts and a few Shoulds from our third feature. Again, we focus on our Musts before working on the Shoulds or Coulds.

Of course, we can always reprioritize something if we learn that it is more or less important than we thought before. And if we have worked through all of the Musts for a feature, we’re a few sprints down the calendar, and we learn of something that is a blocker, it’s okay to slip it in at the top of the prioritized list.

At the end of the time we have for the project, all of the features will be in the project in some form, but they might not be polished.

Continual Prioritization

Another approach to flexibility is to continuously prioritize our list of tasks using the binary insertion (or similar) method. As we add new tasks, we place them in our list ahead of anything less important and behind anything more important. As we learn more about our project, we revisit our priorities and shuffle tasks around to match what we’ve learned.

For each sprint, take the top items from the list and work on them. If we don’t work through everything, then anything left is placed
back into the prioritized list as if it were a new task. Then, we take the top few items for the next sprint.

At the end of the time we have for the project, all of the important things will be done because anything not done will be less important than anything we did. Perhaps not all of the features we wanted made it into the project, but the important ones did.
References and Readings

Aha! https://www.aha.io/.


Agile contracts

The **Agile fixed price** is a contractual model agreed upon by suppliers and customers of IT projects that develop software using Agile methods. The model introduces an initial test phase after which budget, due date, and the way of steering the scope within the framework is agreed upon.

This differs from traditional fixed-price contracts in that fixed-price contracts usually require a detailed and exact description of the subject matter of the contract in advance. Fixed price contracts aim at minimizing the potential risk caused by unpredictable, later changes. In contrast, Agile fixed price contracts simply require a broad description of the entire project instead of a detailed one.[1]

In Agile contracts, the supplier and the customer together define their common assumptions in terms of the business value, implementation risks, expenses (effort) and costs. On the basis of these assumptions, an indicative fixed price scope is agreed upon which is not yet contractually binding. This is followed by the test phase (checkpoint phase), during which the actual implementation begins. At the end of this phase, both parties compare the empirical findings with their initial assumptions. Together, they then decide on the implementation of the entire project and fixate the conditions under which changes are allowed to happen.

Further aspects of an Agile contract are risk share (both parties divide the additional expenses for unexpected changes equally amongst themselves) or the option of either party leaving the contract at any stage (exit points).

### Contents

- Approaches to Agile Contracts
  - Capped Time and materials Contracts
  - Target Cost Contracts
  - Incremental Delivery Contracts
- Criticism
- Literature
- References

### Approaches to Agile Contracts

#### Capped Time and materials Contracts

Capped T&M contracts work in the sense of traditional T&M contracts. However, there is an upper limit to how much customers will have to pay. In this way, suppliers benefit with early time-frame changes while customers only have to pay up until the capped cost limit is reached.[2]

#### Target Cost Contracts
In target cost contracts, parties involved with the contracts agree on a final price during negotiation. These contracts allow cost savings for both parties if contracts run below budget, but also allows both parties to be faced with additional costs if contracts run above budget.

**Incremental Delivery Contracts**

Incremental Delivery Contracts allow customers to review contracts at designated points in the contract life cycle. These points are negotiated into contracts and allow customers to make changes, continue, or terminate the project.

- The first step encompasses the content of the contract on a broad level. This includes the most important project goals, topics and epics as well as the legal framework.[3]
- One of the epics that best represents the entire project is chosen, specified in more detail and turned into several user stories. A well-chosen epic should turn into a good number of user stories, all of which are different and include a range of different features. Thus, they can be used as reference user stories.[4]
- Supplier and customer then come together in a workshop to define the expenses for the entire project based on these reference user stories and all other epics with the help of story points. Assumptions are also made in terms of implementation risk and business value. This information then leads to an indicative fixed price framework, which is not yet legally binding and only fixed at a later step (at the end of the so-called checkpoint phase).
- Then the checkpoint phase is defined. It is the test phase for collaboration during which implementation is begun and initial empirical insight is achieved. It is recommended to make this phase last about two to five sprints (at a sprint length of two weeks). At the end of the checkpoint phase, the customer and supplier check their initial assumptions and decide whether they still want to realize the entire or part of the project. Only then, the indicative fixed price framework is formally agreed upon and becomes contractually binding. During the checkpoint phase, the risk share is also determined, which defines the extent to which additional expenses larger than the fixed price will be charged to the customer.[5]
- Furthermore, the roles in charge of steering the project have to be defined and filled. The customer provides the project manager, the supplier the Product Owner. It is recommended to involve an independent IT consultant selected by mutual agreement of both parties. Together, these roles form a steering committee (scope governance[6]) with the mandate of a formal decision-making panel, which meets on a regular basis and ensures that the continuous specification process is adhered to including that the highest prioritized requirements are specified as user stories.[7]
- In contrast to traditional fixed-price projects, projects with an Agile contract can run out early if the customer believes to have gained the expected value through the already delivered features. This might happen before all agreed-upon functionalities have been implemented. In order for this contractual flexibility to be advantageous to customer and supplier alike, specific agreements must be reached i.e. the supplier could be paid a certain percentage of the budget left over for the undelivered functionality or the supplier could receive a new assignment of the scope of the left-over budget.

**Criticism**

The Agile fixed price is a contract framework most suitable for complex IT projects, where scope, progress and costs are difficult to determine in advance. For standard projects, which have already taken place in the same or a similar way in the past, the test phase and the assessment of the project progress may be skipped. In order for this contractual model to be successful, the supplier and customer should collaborate closely throughout the entire length of the project. Furthermore, a certain amount of mutual trust is imperative in order to be able to agree on the budget, expenses and range of features. It is also advisable to ensure that the broad requirements (epics) listed at the beginning of the project are turned into smaller, more detailed requirements (user stories) as soon as possible. Otherwise, the potential for uncertainty and its connected risks rises.[8]

**Literature**

References


This page was last edited on 12 April 2018, at 22:45 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.
Agile software development is an approach to software development under which requirements and solutions evolve through the collaborative effort of self-organizing and cross-functional teams and their customer(s)/end user(s).[1] It advocates adaptive planning, evolutionary development, empirical knowledge, and continual improvement, and it encourages rapid and flexible response to change.[2]

The term agile (sometimes written Agile)[3] was popularized, in this context, by the Manifesto for Agile Software Development.[4] The values and principles espoused in this manifesto were derived from and underpin a broad range of software development frameworks, including Scrum and Kanban.[5][6]

There is significant anecdotal evidence that adopting agile practices and values improves the agility of software professionals, teams and organizations; however, some empirical studies have found no scientific evidence.[7][8]

## Contents

- **History**
  - The Manifesto for Agile Software Development
    - Agile software development values
    - Agile software development principles
  - Overview
    - Iterative, incremental and evolutionary
    - Efficient and face-to-face communication
    - Very short feedback loop and adaptation cycle
    - Quality focus
  - Philosophy
    - Adaptive vs. predictive
    - Agile vs. waterfall
    - Code vs. documentation
  - Agile software development methods
    - Agile software development practices
    - Method tailoring
    - Large-scale, offshore and distributed
    - Regulated domains
  - Experience and adoption
    - Measuring agility
    - Common agile software development pitfalls
  - Agile management
    - Applications outside software development
  - Criticism
  - See also
  - References
  - Further reading
  - External links
Iterative and incremental development methods can be traced back as early as 1957, with evolutionary project management and adaptive software development emerging in the early 1970s. During the 1990s, a number of lightweight software development methods evolved in reaction to the prevailing heavyweight methods that critics described as overly regulated, planned, and micro-managed. These included: rapid application development (RAD), from 1991; the unified process (UP) and dynamic systems development method (DSDM), both from 1994; Scrum, from 1995; Crystal Clear and extreme programming (XP), both from 1996; and feature-driven development, from 1997. Although these all originated before the publication of the Agile Manifesto, they are now collectively referred to as agile software development methods. At the same time, similar changes were underway in manufacturing and aerospace.

In 2001, seventeen software developers met at a resort in Snowbird, Utah to discuss these lightweight development methods, including among others Kent Beck, Ward Cunningham, Dave Thomas, Jeff Sutherland, Ken Schwaber, Jim Highsmith, Alistair Cockburn, and Robert C. Martin. Together they published the Manifesto for Agile Software Development.

In 2005, a group headed by Cockburn and Highsmith wrote an addendum of project management principles, the PM Declaration of Interdependence, to guide software project management according to agile software development methods.

In 2009, a group working with Martin wrote an extension of software development principles, the Software Craftsmanship Manifesto, to guide agile software development according to professional conduct and mastery.

In 2011, the Agile Alliance created the Guide to Agile Practices (renamed the Agile Glossary in 2016), an evolving open-source compendium of the working definitions of agile practices, terms, and elements, along with interpretations and experience guidelines from the worldwide community of agile practitioners.

The Manifesto for Agile Software Development

Agile software development values

Based on their combined experience of developing software and helping others do that, the seventeen signatories to the manifesto proclaimed that they value:

- **Individuals and Interactions** over processes and tools
- **Working Software** over comprehensive documentation
- **Customer Collaboration** over contract negotiation
- **Responding to Change** over following a plan

That is to say, the items on the left are valued more than the items on the right.

As Scott Ambler elucidated:

- Tools and processes are important, but it is more important to have competent people working together effectively.
- Good documentation is useful in helping people to understand how the software is built and how to use it, but the main point of development is to create software, not documentation.
- A contract is important but is no substitute for working closely with customers to discover what they need.
A project plan is important, but it must not be too rigid to accommodate changes in technology or the environment, stakeholders' priorities, and people's understanding of the problem and its solution.

Some of the authors formed the Agile Alliance, a non-profit organization that promotes software development according to the manifesto's values and principles. Introducing the manifesto on behalf of the Agile Alliance, Jim Highsmith said,

> The Agile movement is not anti-methodology, in fact many of us want to restore credibility to the word methodology. We want to restore a balance. We embrace modeling, but not in order to file some diagram in a dusty corporate repository. We embrace documentation, but not hundreds of pages of never-maintained and rarely-used tomes. We plan, but recognize the limits of planning in a turbulent environment. Those who would brand proponents of XP or SCRUM or any of the other Agile Methodologies as "hackers" are ignorant of both the methodologies and the original definition of the term hacker.

— Jim Highsmith, History: The Agile Manifesto[20]

### Agile software development principles

The **Manifesto for Agile Software Development** is based on twelve principles:[21]

1. Customer satisfaction by early and continuous delivery of valuable software.
2. Welcome changing requirements, even in late development.
3. Deliver working software frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the primary measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. Best architectures, requirements, and designs emerge from self-organizing teams
12. Regularly, the team reflects on how to become more effective, and adjusts accordingly

### Overview

**Iterative, incremental and evolutionary**

Most agile development methods break product development work into small increments that minimize the amount of up-front planning and design. Iterations, or sprints, are short time frames (timeboxes) that typically last from one to four weeks. Each iteration involves a cross-functional team working in all functions: planning, analysis, design, coding, unit testing, and acceptance testing. At the end of the iteration a working product is demonstrated to stakeholders. This minimizes overall risk and allows the product to adapt to changes quickly.[22] An iteration might not add enough functionality to warrant a market release, but the goal is to have an available release (with minimal bugs) at the end of each iteration.[23] Multiple iterations might be required to release a product or new features. Working software is the primary measure of progress.[21]
Efficient and face-to-face communication

The principle of co-location is that co-workers on the same team should be situated together to better establish the identity as a team and to improve communication.[24] This enables face-to-face interaction, ideally in front of a whiteboard, that reduces the cycle time typically taken when questions and answers are mediated through phone, persistent chat, wiki, or email.[25]

No matter which development method is followed, every team should include a customer representative ("Product Owner" in Scrum). This person is agreed by stakeholders to act on their behalf and makes a personal commitment to being available for developers to answer questions throughout the iteration. At the end of each iteration, stakeholders and the customer representative review progress and re-evaluate priorities with a view to optimizing the return on investment (ROI) and ensuring alignment with customer needs and company goals.

In agile software development, an information radiator is a (normally large) physical display located prominently near the development team, where passers-by can see it. It presents an up-to-date summary of the product development status.[26][27] A build light indicator may also be used to inform a team about the current status of their product development.

Very short feedback loop and adaptation cycle

A common characteristic in agile software development is the daily stand-up (also known as the daily scrum). In a brief session, team members report to each other what they did the previous day toward their team's iteration goal, what they intend to do today toward the goal, and any roadblocks or impediments they can see to the goal.[28]

Quality focus

Specific tools and techniques, such as continuous integration, automated unit testing, pair programming, test-driven development, design patterns, behavior-driven development, domain-driven design, code refactoring and other techniques are often used to improve quality and enhance product development agility.[29] This is predicated on designing and building quality in from the beginning and being able to demonstrate software for customers at any point, or at least at the end of every iteration.[30]

Philosophy

Compared to traditional software engineering, agile software development mainly targets complex systems and product development with dynamic, non-deterministic and non-linear characteristics. Accurate estimates, stable plans, and predictions are often hard to get in early stages, and confidence in them is likely to be low. Agile practitioners will seek to reduce the leap-of-faith that is needed before any evidence of value can be obtained.[31] Requirements and design are held to be emergent. Big up-front specifications would probably cause a lot of waste in such cases, i.e., are not economically sound. These basic arguments and previous industry experiences, learned from years of successes and failures, have helped shape agile development's favor of adaptive, iterative and evolutionary development.[32]

Adaptive vs. predictive

Development methods exist on a continuum from adaptive to predictive.[33] Agile software development methods lie on the adaptive side of this continuum. One key of adaptive development methods is a rolling wave approach to schedule planning, which identifies milestones but leaves flexibility in the path to reach them, and also allows for the milestones themselves to change.[34]
Adaptive methods focus on adapting quickly to changing realities. When the needs of a project change, an adaptive team changes as well. An adaptive team has difficulty describing exactly what will happen in the future. The further away a date is, the more vague an adaptive method is about what will happen on that date. An adaptive team cannot report exactly what tasks they will do next week, but only which features they plan for next month. When asked about a release six months from now, an adaptive team might be able to report only the mission statement for the release, or a statement of expected value vs. cost.

Predictive methods, in contrast, focus on analysing and planning the future in detail and cater for known risks. In the extremes, a predictive team can report exactly what features and tasks are planned for the entire length of the development process. Predictive methods rely on effective early phase analysis and if this goes very wrong, the project may have difficulty changing direction. Predictive teams often institute a change control board to ensure they consider only the most valuable changes.

Risk analysis can be used to choose between adaptive (agile or value-driven) and predictive (plan-driven) methods. Barry Boehm and Richard Turner suggest that each side of the continuum has its own home ground, as follows:

<table>
<thead>
<tr>
<th>Home grounds of different development methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Value-driven methods</strong></td>
</tr>
<tr>
<td>Low criticality</td>
</tr>
<tr>
<td>Senior developers</td>
</tr>
<tr>
<td>Requirements change often</td>
</tr>
<tr>
<td>Small number of developers</td>
</tr>
<tr>
<td>Culture that responds to change</td>
</tr>
<tr>
<td><strong>Plan-driven methods</strong></td>
</tr>
<tr>
<td>High criticality</td>
</tr>
<tr>
<td>Junior developers(?)</td>
</tr>
<tr>
<td>Requirements do not change often</td>
</tr>
<tr>
<td>Large number of developers</td>
</tr>
<tr>
<td>Culture that demands order</td>
</tr>
<tr>
<td><strong>Formal methods</strong></td>
</tr>
<tr>
<td>Extreme criticality</td>
</tr>
<tr>
<td>Senior developers</td>
</tr>
<tr>
<td>Limited requirements, limited features</td>
</tr>
<tr>
<td>see Wirth’s law</td>
</tr>
<tr>
<td>Requirements that can be modeled</td>
</tr>
<tr>
<td>Extreme quality</td>
</tr>
</tbody>
</table>

**Agile vs. waterfall**

One of the differences between agile software development methods and waterfall is the approach to quality and testing. In the waterfall model, there is always a separate testing phase after a build phase; however, in agile software development testing is completed in the same iteration as programming.

Because testing is done in every iteration—which develops a small piece of the software—users can frequently use those new pieces of software and validate the value. After the users know the real value of the updated piece of software, they can make better decisions about the software's future. Having a value retrospective and software re-planning session in each iteration—Scrum typically has iterations of just two weeks—helps the team continuously adapt its plans so as to maximize the value it delivers. This follows a pattern similar to the PDCA cycle, as the work is planned, done, checked (in the review and retrospective), and any changes agreed are acted upon.

This iterative approach supports a product rather than a project mindset. This provides greater flexibility throughout the development process; whereas on projects the requirements are defined and locked down from the very beginning, making it difficult to change them later. Iterative product development allows the software to evolve in response to changes in business environment or market requirements.

Because of the short iteration style of agile software development, it also has strong connections with the lean startup concept.
**Code vs. documentation**

In a letter to *IEEE Computer*, Steven Rakitin expressed cynicism about agile software development, calling it "yet another attempt to undermine the discipline of software engineering" and translating "working software over comprehensive documentation" as "we want to spend all our time coding. Remember, real programmers don’t write documentation.‖[38]

This is disputed by proponents of agile software development, who state that developers should write documentation if that is the best way to achieve the relevant goals, but that there are often better ways to achieve those goals than writing static documentation.[39] Scott Ambler states that documentation should be "just barely good enough" (JBGE),[40] that too much or comprehensive documentation would usually cause waste, and developers rarely trust detailed documentation because it's usually out of sync with code,[39] while too little documentation may also cause problems for maintenance, communication, learning and knowledge sharing. Alistair Cockburn wrote of the *Crystal Clear* method:

> Crystal considers development a series of co-operative games, and intends that the documentation is enough to help the next win at the next game. The work products for Crystal include use cases, risk list, iteration plan, core domain models, and design notes to inform on choices...however there are no templates for these documents and descriptions are necessarily vague, but the objective is clear, **just enough documentation** for the next game. I always tend to characterize this to my team as: what would you want to know if you joined the team tomorrow.

— Alistair Cockburn.[41]

**Agile software development methods**

Agile software development methods support a broad range of the software development life cycle.[42] Some focus on the practices (e.g., XP, pragmatic programming, agile modeling), while some focus on managing the flow of work (e.g., Scrum, Kanban). Some support activities for requirements specification and development (e.g., FDD), while some seek to cover the full development life cycle (e.g., DSDM, RUP).

Popular agile software development frameworks include (but are not limited to):

- Adaptive software development (ASD)
- Agile modeling
- Agile unified process (AUP)
- Disciplined agile delivery
- Dynamic systems development method (DSDM)
- Extreme programming (XP)
- Feature-driven development (FDD)
- Lean software development
- Kanban
- Rapid application development (RAD)
- Scrum
- Scrumban

**Agile software development practices**
Agile software development is supported by a number of concrete practices, covering areas like requirements, design, modeling, coding, testing, planning, risk management, process, quality, etc. Some notable agile software development practices include:

- Acceptance test-driven development (ATDD)
- Agile modeling
- Agile testing
- Backlogs (Product and Sprint)
- Behavior-driven development (BDD)
- Business analyst designer method (BADM)
- Continuous integration (CI)
- Cross-functional team
- Domain-driven design (DDD)
- Information radiators (scrum board, task board, visual management board, burndown chart)
- Iterative and incremental development (IID)
- Low-code development platforms
- Pair programming
- Planning poker
- Refactoring
- Retrospective
- Scrum events (sprint planning, daily scrum, sprint review and retrospective)
- Story-driven modeling
- Test-driven development (TDD)
- Timeboxing
- User story
- User story mapping
- Velocity tracking

Method tailoring

In the literature, different terms refer to the notion of method adaptation, including 'method tailoring', 'method fragment adaptation' and 'situational method engineering'. Method tailoring is defined as:

A process or capability in which human agents determine a system development approach for a specific project situation through responsive changes in, and dynamic interplays between contexts, intentions, and method fragments.

— Mehmet Nafiz Aydin et al., An Agile Information Systems Development Method in use

Situation-appropriateness should be considered as a distinguishing characteristic between agile methods and more plan-driven software development methods, with agile methods allowing product development teams to adapt working practices according to the needs of individual products. Potentially, most agile methods could be suitable for method tailoring, such as DSDM tailored in a CMM context and XP tailored with the Rule Description Practices (RDP) technique. Not all agile proponents agree, however, with Schwaber noting "that is how we got into trouble in the first place, thinking that the problem was not having a perfect methodology. Efforts [should] center on the changes [needed] in the enterprise". Bas Vodde reinforced this viewpoint, suggesting that unlike traditional, large methodologies that require you to pick and choose elements, Scrum provides the basics on top of which you add additional
elements to localise and contextualise its use. Practitioners seldom use system development methods, or agile methods specifically, by the book, often choosing to omit or tailor some of the practices of a method in order to create an in-house method.

In practice, methods can be tailored using various tools. Generic process modeling languages such as Unified Modeling Language can be used to tailor software development methods. However, dedicated tools for method engineering such as the Essence Theory of Software Engineering of SEMAT also exist.

Large-scale, offshore and distributed

Agile software development has been widely seen as highly suited to certain types of environments, including small teams of experts working on greenfield projects and the challenges and limitations encountered in the adoption of agile software development methods in a large organization with legacy infrastructure are well-documented and understood.

In response, a range of strategies and patterns has evolved for overcoming challenges with large-scale development efforts (>20 developers) or distributed (non-colocated) development teams amongst other challenges; and there are now several recognised frameworks that seek to mitigate or avoid these challenges.

- Scaled agile framework (SAFe), Dean Leffingwell
- Disciplined agile delivery (DAD), Scott Ambler
- Large-scale scrum (LeSS), Craig Larman and Bas Vodde
- Nexus (scaled professional Scrum), Ken Schwaber
- Scrum at Scale, Jeff Sutherland, Alex Brown
- Enterprise Scrum, Mike Beedle
- Setchu (Scrum-based lightweight framework), Michael Ebbage
- Xscale
- Agile path
- Holistic Software Development

There are many conflicting viewpoints on whether all of these are effective or indeed fit the definition of agile development, and this remains an active and ongoing area of research.

When agile software development is applied in a distributed setting (with teams dispersed across multiple business locations), it is commonly referred to as distributed agile development. The goal is to leverage the unique benefits offered by each approach. Distributed development allow organizations to build software by strategically setting up teams in different parts of the globe, virtually building software round-the-clock (more commonly referred to as follow-the-sun model). On the other hand, agile development provides increased transparency, continuous feedback and more flexibility when responding to changes.

Regulated domains

Agile software development methods were initially seen as best suitable for non-critical product developments, thereby excluded from use in regulated domains such as medical devices, pharmaceutical, financial, nuclear systems, automotive, and avionics sectors, etc. However, in the last several years, there have been several initiatives for the adaptation of agile methods for these domains.

There are numerous standards that may apply in regulated domains, including ISO 26262, ISO 9000, ISO 9001, and ISO/IEC 15504. A number of key concerns are of particular importance in regulated domains:
- Quality assurance (QA): Systematic and inherent quality management underpinning a controlled professional process and reliability and correctness of product.
- Safety and security: Formal planning and risk management to mitigate safety risks for users and securely protecting users from unintentional and malicious misuse.
- Traceability: Documentation providing auditable evidence of regulatory compliance and facilitating traceability and investigation of problems.
- Verification and Validation (V&V): Embedded throughout the software development process (e.g. user requirements specification, functional specification, design specification, code review, unit tests, integration tests, system tests).

**Experience and adoption**

Although agile software development methods can be used with any programming paradigm or language in practice, they were originally closely associated with object-oriented environments such as Smalltalk and Lisp and later Java. The initial adopters of agile methods were usually small to medium-sized teams working on unprecedented systems with requirements that were difficult to finalize and likely to change as the system was being developed. This section describes common problems that organizations encounter when they try to adopt agile software development methods as well as various techniques to measure the quality and performance of agile teams.[75]

**Measuring agility**

The best agile practitioners have always emphasized thorough engineering principles. As a result, there are a number of best practices and tools for measuring the performance of agile software development and teams.

**Internal assessments**

The Agility measurement index, amongst others, rates developments against five dimensions of product development (duration, risk, novelty, effort, and interaction).[76][77] Other techniques are based on measurable goals[78] and one study suggests that velocity can be used as a metric of agility.[79] There are also agile self-assessments to determine whether a team is using agile software development practices (Nokia test,[80] Karlskrona test,[81] 42 points test).[82]

**Public surveys**

One of the early studies reporting gains in quality, productivity, and business satisfaction by using agile software developments methods was a survey conducted by Shine Technologies from November 2002 to January 2003.[83]

A similar survey, the State of Agile, is conducted every year starting in 2006 with thousands of participants from around the software development community. This tracks trends on the benefits of agility, lessons learned, and good practices. Each survey has reported increasing numbers saying that agile software development helps them deliver software faster; improves their ability to manage changing customer priorities; and increases their productivity.[84] Surveys have also consistently shown better results with agile product development methods compared to classical project management.[85][86] In balance, there are reports that some feel that agile development methods are still too young to enable extensive academic research of their success.[87]

**Common agile software development pitfalls**

Organizations and teams implementing agile software development often face difficulties transitioning from more traditional methods such as waterfall development, such as teams having an agile process forced on them.[88] These are often termed agile anti-patterns or more commonly agile smells. Below are some common examples:
Lack of overall product design

A goal of agile software development is to focus more on producing working software and less on documentation. This is in contrast to waterfall models where the process is often highly controlled and minor changes to the system require significant revision of supporting documentation. However, this does not justify completely doing without any analysis or design at all. Failure to pay attention to design can cause a team to proceed rapidly at first but then to have significant rework required as they attempt to scale up the system. One of the key features of agile software development is that it is iterative. When done correctly design emerges as the system is developed and commonalities and opportunities for re-use are discovered.\[89\]

Adding stories to an iteration in progress

In agile software development, stories (similar to use case descriptions) are typically used to define requirements and an iteration is a short period of time during which the team commits to specific goals.\[90\] Adding stories to an iteration in progress is detrimental to a good flow of work. These should be added to the product backlog and prioritized for a subsequent iteration or in rare cases the iteration could be cancelled.\[91\]

This does not mean that a story cannot expand. Teams must deal with new information, which may produce additional tasks for a story. If the new information prevents the story from being completed during the iteration, then it should be carried over to a subsequent iteration. However, it should be prioritized against all remaining stories, as the new information may have changed the story's original priority.

Lack of sponsor support

Agile software development is often implemented as a grassroots effort in organizations by software development teams trying to optimize their development processes and ensure consistency in the software development life cycle. By not having sponsor support, teams may face difficulties and resistance from business partners, other development teams and management. Additionally, they may suffer without appropriate funding and resources.\[92\] This increases the likelihood of failure.\[93\]

Insufficient training

A survey performed by VersionOne found respondents cited insufficient training as the most significant cause for failed agile implementations.\[94\] Teams have fallen into the trap of assuming the reduced processes of agile software development compared to other methodologies such as waterfall means that there are no actual rules for agile software development.

Product owner role is not properly filled

The product owner is responsible for representing the business in the development activity and is often the most demanding role.\[95\]

A common mistake is to have the product owner role filled by someone from the development team. This requires the team to make its own decisions on prioritization without real feedback from the business. They try to solve business issues internally or delay work as they reach outside the team for direction. This often leads to distraction and a breakdown in collaboration.\[96\]

Teams are not focused
Agile software development requires teams to meet product commitments, which means they should focus only on work for that product. However, team members who appear to have spare capacity are often expected to take on other work, which makes it difficult for them to help complete the work to which their team had committed.[97]

**Excessive preparation/planning**

Teams may fall into the trap of spending too much time preparing or planning. This is a common trap for teams less familiar with agile software development where the teams feel obliged to have a complete understanding and specification of all stories. Teams should be prepared to move forward only with those stories in which they have confidence, then during the iteration continue to discover and prepare work for subsequent iterations (often referred to as backlog refinement or grooming).

**Problem-solving in the daily standup**

A daily standup should be a focused, timely meeting where all team members disseminate information. If problem-solving occurs, it often can only involve certain team members and potentially is not the best use of the entire team’s time. If during the daily standup the team starts diving into problem-solving, it should be set aside until a sub-team can discuss, usually immediately after the standup completes. [98]

**Assigning tasks**

One of the intended benefits of agile software development is to empower the team to make choices, as they are closest to the problem. Additionally, they should make choices as close to implementation as possible, to use more timely information in the decision. If team members are assigned tasks by others or too early in the process, the benefits of localized and timely decision making can be lost.[99]

Being assigned work also constrains team members into certain roles (for example, team member A must always do the database work), which limits opportunities for cross-training.[99] Team members themselves can choose to take on tasks that stretch their abilities and provide cross-training opportunities.

**Scrum master as a contributor**

Another common pitfall is for a scrum master to act as a contributor. While not prohibited by the Scrum methodology, the scrum master needs to ensure they have the capacity to act in the role of scrum master first and not working on development tasks. A scrum master’s role is to facilitate the process rather than create the product.[100]

Having the scrum master also multitasking may result in too many context switches to be productive. Additionally, as a scrum master is responsible for ensuring roadblocks are removed so that the team can make forward progress, the benefit gained by individual tasks moving forward may not outweigh roadblocks that are deferred due to lack of capacity.[100]

**Lack of test automation**

Due to the iterative nature of agile development, multiple rounds of testing are often needed. Automated testing helps reduce the impact of repeated unit, integration, and regression tests and frees developers and testers to focus on higher value work.[101]

Test automation also supports continued refactoring required by iterative software development. Allowing a developer to quickly run tests to confirm refactoring has not modified the functionality of the application may reduce the workload and increase confidence that cleanup efforts have not introduced new defects.
Allowing technical debt to build up

Focusing on delivering new functionality may result in increased technical debt. The team must allow themselves time for defect remediation and refactoring. Technical debt hinders planning abilities by increasing the amount of unscheduled work as production defects distract the team from further progress.\footnote{102}

As the system evolves it is important to refactor as entropy of the system naturally increases.\footnote{103} Over time the lack of constant maintenance causes increasing defects and development costs.\footnote{102}

Attempting to take on too much in an iteration

A common misconception is that agile software development allows continuous change, however an iteration backlog is an agreement of what work can be completed during an iteration.\footnote{104} Having too much work-in-progress (WIP) results in inefficiencies such as context-switching and queueing.\footnote{105} The team must avoid feeling pressured into taking on additional work.\footnote{106}

Fixed time, resources, scope, and quality

Agile software development fixes time (iteration duration), quality, and ideally resources in advance (though maintaining fixed resources may be difficult if developers are often pulled away from tasks to handle production incidents), while the scope remains variable. The customer or product owner often push for a fixed scope for an iteration. However, teams should be reluctant to commit to the locked time, resources and scope (commonly known as the project management triangle). Efforts to add scope to the fixed time and resources of agile software development may result in decreased quality.\footnote{107}

Developer burnout

Due to the focused pace and continuous nature of agile practices, there is a heightened risk of burnout among members of the delivery team.\footnote{108}

Agile management

The term \textit{agile management} is applied to an iterative, incremental method of managing the design and build activities of engineering, information technology and other business areas that aim to provide new product or service development in a highly flexible and interactive manner, based on the principles expressed in the \textit{Manifesto for Agile Software Development}.\footnote{109}

Agile X techniques may also be called \textit{extreme project management}. It is a variant of iterative life cycle\footnote{110} where deliverables are submitted in stages. The main difference between agile and iterative development is that agile methods complete small portions of the deliverables in each delivery cycle (iteration)\footnote{111} while iterative methods evolve the entire set of deliverables over time, completing them near the end of the project. Both iterative and agile methods were developed as a reaction to various obstacles that developed in more sequential forms of project organization. For example, as technology projects grow in complexity, end users tend to have difficulty defining the long-term requirements without being able to view progressive prototypes. Projects that develop in iterations can constantly gather feedback to help refine those requirements.

Agile management also offers a simple framework promoting communication and reflection on past work amongst team members.\footnote{112} Teams who were using traditional waterfall planning and adopted the agile way of development typically go through a transformation phase and often take help from agile coaches who help guide the teams through a smooth
transformation. There are typically two styles of agile coaching: push-based and pull-based agile coaching. Agile management approaches have also been employed and adapted to the business and government sectors. For example, within the federal government of the United States, the United States Agency for International Development (USAID) is employing a collaborative project management approach that focuses on incorporating collaborating, learning and adapting (CLA) strategies to iterate and adapt programming.[113]

Agile methods are mentioned in the Guide to the Project Management Body of Knowledge (PMBOK Guide) under the Project Lifecycle definition:

**Adaptive project life cycle**, a project life cycle, also known as change-driven or agile methods, that is intended to facilitate change and require a high degree of ongoing stakeholder involvement. Adaptive life cycles are also iterative and incremental, but differ in that iterations are very rapid (usually 2-4 weeks in length) and are fixed in time and resources.[114]

Applications outside software development

According to Jean-Loup Richet (Research Fellow at ESSEC Institute for Strategic Innovation & Services) "this approach can be leveraged effectively for non-software products and for project management in general, especially in areas of innovation and uncertainty." The end result is a product or project that best meets current customer needs and is delivered with minimal costs, waste, and time, enabling companies to achieve bottom line gains earlier than via traditional approaches.[115]

Agile software development methods have been extensively used for development of software products and some of them use certain characteristics of software, such as object technologies.[116] However, these techniques can be applied to the development of non-software products, such as computers, motor vehicles,[117] medical devices, food, clothing, and music.[118] Agile software development methods have been used in non-development IT infrastructure deployments and migrations. Some of the wider principles of agile software development have also found application in general management[119] (e.g., strategy, governance, risk, finance) under the terms business agility or agile business management.

Under an agile business management model, agile software development techniques, practices, principles and values are expressed across five domains.[120]

1. Integrated customer engagement: to embed customers within any delivery process to share accountability for product/service delivery.
2. Facilitation-based management: adopting agile management models, like the role of Scrum Master, to facilitate the day-to-day operation of teams.
3. Agile work practices: adopting specific iterative and incremental work practices such as Scrum, Kanban, test-driven development or feature-driven development across all business functions (from sales, human resources, finance[121] and marketing).
4. An enabling organisational structure: with a focus on staff engagement, personal autonomy and outcomes based governance.
5. Applications of agile process (along with DevOps and lean manufacturing), to data analytics, business intelligence, big data, and data science is called DataOps
Agile software development paradigms can be used in other areas of life such as raising children. Its success in child development might be founded on some basic management principles; communication, adaptation, and awareness. In a TED Talk, Bruce Feiler shared how he applied basic agile paradigms to household management and raising children.[122]

**Criticism**

Agile practices can be inefficient in large organizations and certain types of developments.[123] Many organizations believe that agile software development methodologies are too extreme and adopt a Hybrid approach [124] that mixes elements of agile software development and plan-driven approaches.[125] Some methods, such as dynamic systems development method (DSDM) attempt this in a disciplined way, without sacrificing fundamental principles.

The increasing adoption of agile practices has also been criticized as being a management fad that simply describes existing good practices under new jargon, promotes a one size fits all mindset towards development strategies, and wrongly emphasizes method over results.[126]

Alistair Cockburn organized a celebration of the 10th anniversary of the Manifesto for Agile Software Development in Snowbird, Utah on 12 February 2011, gathering some 30+ people who had been involved at the original meeting and since. A list of about 20 elephants in the room ('undiscussable' agile topics/issues) were collected, including aspects: the alliances, failures and limitations of agile software development practices and context (possible causes: commercial interests, decontextualization, no obvious way to make progress based on failure, limited objective evidence, cognitive biases and reasoning fallacies), politics and culture.[127] As Philippe Kruchten wrote:

> The agile movement is in some ways a bit like a teenager: very self-conscious, checking constantly its appearance in a mirror, accepting few criticisms, only interested in being with its peers, rejecting en bloc all wisdom from the past, just because it is from the past, adopting fads and new jargon, at times cocky and arrogant. But I have no doubts that it will mature further, become more open to the outside world, more reflective, and therefore, more effective.

— Philippe Kruchten[127]

**See also**

- Workers' self-management

**References**

4. Kent Beck; James Grenning; Robert C. Martin; Mike Beedle; Jim Highsmith; Steve Mellor; Arie van Bennekum; Andrew Hunt; Ken Schwaber; Alistair Cockburn; Ron Jeffries; Jeff Sutherland; Ward Cunningham; Jon Kern; Dave Thomas; Martin Fowler; Brian Marick (2001). "Manifesto for Agile Software Development" (http://agilemanifesto.org/). Agile Alliance. Retrieved 14 June 2010.
5. Which is better – Kanban or Scrum? (http://www.cleverpm.com/2016/03/04/which-is-better-kanban-or-scrum/)


38. Rakitin, Steven R. (2001). "Manifesto Elicits Cynicism: Reader's letter to the editor by Steven R. Rakitin". *IEEE Computer*. 34: 4. "The article titled 'Agile Software Development: The Business of Innovation' ... is yet another attempt to undermine the discipline of software engineering ... We want to spend all our time coding. Remember, real programmers don't write documentation."


http://agiletng.org/2014/04/21/xscale/

http://www.agile-path.com/

http://www.holistic-software.com


72. http://www.sintef.no/safescrum


83. "Agile Methodologies Survey Results" (https://web.archive.org/web/20100821225423/http://www.shinetech.com/attachments/104_ShineTechAgileSurvey2003-01-17.pdf) (PDF). Shine Technologies. January 2003. Archived from the original (http://www.shinetech.com/attachments/104_ShineTechAgileSurvey2003-01-17.pdf) (PDF) on 21 August 2010. Retrieved 3 June 2010. "95% stated that there was either no effect or a cost reduction ... 93% stated that productivity was better or significantly better ... 88% stated that quality was better or significantly better ... 83% stated that business satisfaction was better or significantly better"


86. Ambler, Scott (3 August 2006). "Survey Says: Agile Works in Practice" (http://www.drdobbs.com/architecture-and-design/191800169?sessionid=2QJ23QRYM3H4PQE1GHPCKH4ATMY32JVN?queryText=agile+survey). Dr. Dobb's. Retrieved 3 June 2010. "Only 6% indicated that their productivity was lowered ... No change in productivity was reported by 34% of respondents and 60% reported increased productivity ... 66% [responded] that the quality is higher ... 58% of organizations report improved satisfaction, whereas only 3% report reduced satisfaction."


88. Shore & Warden 2008, p. 47


Further reading

- Kupersmith, Kupe. "Agile is a Fad" (http://www.batimes.com/kupe-kupersmith/agile-is-a-fad.html).
External links

- Agile Manifesto (http://agilemanifesto.org/)
- Agile Glossary (https://www.agilealliance.org/agile101/agile-glossary/)
- The New Methodology (http://martinfowler.com/articles/newMethodology.html) Martin Fowler's description of the background to agile methods
- Ten Authors of The Agile Manifesto Celebrate its Tenth Anniversary (http://www.pragprog.com/magazines/2011-02/agile--)
- AgilePatterns.org (http://agilepatterns.org/)


This page was last edited on 1 March 2019, at 13:05 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.
## Comparison of Scrum software

This page compares software with specific support for the Scrum framework. Although the features of some general project management software can be conceptualized around Scrum, general project management software is not included on this list unless it has, or a plugin for it has, specific support for Scrum.

### Contents

- General information
- Sprint features
- Story features
- Task features
- Integration features
- See also
- Notes and references

### General information

<table>
<thead>
<tr>
<th>Software</th>
<th>Web-based</th>
<th>Hosted On-Premises</th>
<th>SaaS</th>
<th>License</th>
<th>Implementation programming languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azure Boards</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Proprietary</td>
<td>.NET and modern Web frameworks</td>
</tr>
<tr>
<td>HeySpace</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Proprietary</td>
<td>JavaScript</td>
</tr>
<tr>
<td>JIRA</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Proprietary</td>
<td>Java</td>
</tr>
<tr>
<td>Projektron BCS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Proprietary</td>
<td>Java</td>
</tr>
<tr>
<td>Taiga</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>GPL</td>
<td>Python and JavaScript</td>
</tr>
<tr>
<td>Trello</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Proprietary</td>
<td>PHP and JavaScript</td>
</tr>
<tr>
<td>Tuleap</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>GPL</td>
<td>PHP and JavaScript</td>
</tr>
</tbody>
</table>

### Sprint features

<table>
<thead>
<tr>
<th>Software</th>
<th>Scrum board</th>
<th>Custom columns</th>
<th>Burndown chart</th>
<th>Sprint goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azure Boards</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Sprint name</td>
</tr>
<tr>
<td>HeySpace</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Sprint description</td>
</tr>
<tr>
<td>JIRA</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Projektron BCS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Taiga</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Trello</td>
<td>Yes</td>
<td>Yes</td>
<td>Via plugin</td>
<td>No</td>
</tr>
<tr>
<td>Tuleap</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Sprint description</td>
</tr>
</tbody>
</table>

### Story features
<table>
<thead>
<tr>
<th>Software</th>
<th>Reorderable product backlog</th>
<th>Story description</th>
<th>Acceptance criteria</th>
<th>Story points</th>
<th>Planning poker</th>
<th>Planning poker points averaging</th>
<th>Story claiming</th>
<th>Comments on stories</th>
<th>Change tracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azure Boards</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Via extension[1]</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>HeySpace</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JIRA</td>
<td>Yes</td>
<td>Yes</td>
<td>Via custom field</td>
<td>Yes</td>
<td>Via plugin</td>
<td>Via plugin</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Projektron BCS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Taiga</td>
<td>Yes</td>
<td>Yes</td>
<td>Via custom field</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Trello</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Via plugin</td>
<td>Via plugin</td>
<td>Via plugin</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Tuleap</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Task features**

<table>
<thead>
<tr>
<th>Software</th>
<th>Tasks within stories</th>
<th>Task claiming</th>
<th>Multiple people can claim tasks</th>
<th>Separate comments on tasks</th>
<th>Auto-story completion[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azure Boards</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Via extension[3]</td>
</tr>
<tr>
<td>HeySpace</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>JIRA</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Optional</td>
</tr>
<tr>
<td>Projektron BCS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Taiga</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Trello</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Tuleap</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Integration features**

<table>
<thead>
<tr>
<th>Software</th>
<th>Slack integration</th>
<th>HipChat integration</th>
<th>Data export</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azure Boards</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>HeySpace</td>
<td>own built-in feature</td>
<td>own built-in feature</td>
<td>No</td>
</tr>
<tr>
<td>JIRA</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Projektron BCS</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Taiga</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Trello</td>
<td>Yes</td>
<td>Yes[4]</td>
<td>Yes</td>
</tr>
<tr>
<td>Tuleap</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also**

- Comparison of project management software
- Kanban (development)

**Notes and references**

2. when all tasks have been marked completed

Fibonacci scale (agile)

In Agile software development, the Fibonacci scale consists of a sequence of numbers used for estimating the relative size of user stories in points. Agile Scrum is based on the concept of working iteratively in short sprints, typically two weeks long, where the requirements and development are continuously being improved. The Fibonacci sequence consists of numbers that are the summation of the two preceding numbers, starting with \[0, 1\]. Agile uses the Fibonacci sequence to achieve better results by reducing complexity, effort, and doubt when determining the development time required for a task, which can range from a few minutes to several weeks.\[1\]

**Contents**

*Procedure*
*Significance*
*Other estimation scales*
*See also*
*References*

**Procedure**

It is easier to determine the relative complexity of a task rather than figuring out how much time it requires. Due to this, when working with agile, a revised Fibonacci scale is used in terms of points to estimate the work, as opposed to the traditional measurement of time.\[2\]

In one method commonly used to calculate the size of stories in points, a process like the game of Planning Poker, the following process is used:

1. The Product Owner sits down with the team to estimate the user stories.
2. Each member estimates a number on the Fibonacci scale that he/she believes represents the size of the task.
3. All members are ordered to disclose their number at once.
4. Any differences in the numbers will be followed by a discussion until a consensus is reached.
5. Each user story is added to a bucket which represents the corresponding point on the Fibonacci scale.
6. The steps above are repeated for all user stories.
7. The buckets are added to the backlog.

Giving each member the opportunity to think individually reduces pressure and may result in a more accurate representation of the feature’s size.

In another commonly used method, a process like the game of Two Pass Relative Sizing\[3\], also known as the Steve Bockman Method\[4\] and the Team Estimation Game\[5\], the following process is used:

1. The product manager sits down with the team to estimate a project’s worth of user stories, which are provided to the team in a stack of 3x5 or 4x6 cards.
2. The first team member reads the first card and places it on the table, passing the remaining stack to the next team member.
3. The second team member reads the second card, may declare a belief that the story is larger or smaller than the card already on the table or may ask the team for help in so determining, and establishes which direction is smaller or
larger by placing the card to the left or right of the first card; and passes the remaining stack to the next team member.

4. The third team member has a choice: to move the position of the second card; or to read the third card, declare a belief that the story is larger than the first two, smaller than the first two; or belongs between the first two; and passes the remaining stack to the next team member.

5. When all the cards are on the table - if the product manager truly provided a project's worth, then there are likely 60 or 100 or 130 cards and the team has had to "snake" them to fit them all - then the team begins with the smallest story and assigns it a "1", continuing to assign "1"s to subsequent stories until they see an obvious jump to "2"s, and then to "3"s, "5"s, "8"s, and so on. With the result that the "snake" of cards is now numbered from the smallest story, a "1", to the largest epic, a "100".

This method has the advantages that numbers are not used until the second pass; that no guessing is required for the first story read as to how big a "5" or "8" or "3" is\[4\]; that stories are truly ordered and numbered relative to each other; and when not everyone can estimate the entire story\[6\].

Regardless of method, as the team goes through multiple sprints and the estimation process is improved, the product manager will be able to determine a stable velocity. The velocity is determined by calculating the number of story points completed in each iteration.\[1\]

**Significance**

People estimate user stories with smaller points more accurately than user stories that have higher costs associated with them. As the numbers increase, the difference between two succeeding numbers increases exponentially and leads to less accurate estimates.\[7\]

- Using Fibonacci series is helpful in this scenario because the larger user stories (i.e. stories of size greater than 8) that tend to lead to inconsistent estimates between each team member can be grouped to the nearest estimated Fibonacci number of the corresponding bucket in the backlog.
- In case of small user stories the bucket difference is small and hence the final cost of resource and time can be finalized more accurately.

One good way to estimate the cost is to express it in terms of multiples of other known user stories' costs. This way, it will be easier for each team member to estimate the relative cost. The estimation process of comparing a story with two previously estimated user stories is called triangulation.\[7\]

The product manager can include a "0" value in the scale indicating that the user stories need very little time or resources.\[7\] However, the user story that was assigned 0 cost cannot be used as a relative scale to estimate the cost of other user stories (i.e. we cannot say a story is 10 times harder than a story of size 0).

An advantage of Fibonacci sequence is that it allows developers to disaggregate a user story from one large bucket into two preceding buckets (since a bucket is formed by adding the size of two preceding buckets).\[7\] This process helps to create optimal user stories.

**Other estimation scales**

- Linear Scale – Increments in a fixed value
- T-shirt Size – (S < M < L < XL < XXL < XXXL)
- Playing Cards – Mostly used in planning poker (A < 2 < 3…)
- Exponential series - \((a^n)\) for some \(a\) and for all integer \(n\geq 0\)

**See also**

- Scrum (software development)
- Software development effort estimation
References


This page was last edited on 6 March 2019, at 22:34 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.
Incremental funding methodology

The Incremental Funding Methodology (IFM) is an ROI-informed approach to software development in which software is developed and delivered in carefully prioritized chunks of customer valued functionality. These chunks are known as Minimum Marketable Features (MMFs).

IFM integrates traditional software engineering activities with financially informed project management strategies. IFM heuristics provide clarity into important metrics such as project level NPV, ROI, initial start-up investment costs, and time needed for a project to reach self-funding status. It enables developers, customers, and business stakeholders to answer critical questions related to the development and delivery of a product and to optimize strategies accordingly.

In short, IFM equips developers and project managers with techniques and principles for increasing the financial returns of a software project and for identifying development schedules that make a project financially feasible.

See also

- Minimum viable product

Further reading

Kanban (development)

Kanban (Japanese 看板, signboard or billboard) is a lean method to manage and improve work across human systems. This approach aims to manage work by balancing demands with available capacity, and by improving the handling of system-level bottlenecks.

Work items are visualized to give participants a view of progress and process, from start to finish - usually via a Kanban board. Work is pulled as capacity permits, rather than work being pushed into the process when requested.

In knowledge work and in software development, the aim is to provide a visual process-management system which aids decision-making about what, when and how much to produce. The underlying Kanban method originated in lean manufacturing[1] (inspired by the Toyota Production System[2]) it is now used in software development and technology-related work and has been combined with other methods or frameworks such as Scrum.[3]

Overview


Kanban boards

Contents

Overview
Kanban boards
Examples
Software development
Other uses
See also
References
Further reading
Although Kanban does not require that the team or organization use a Kanban board, they can be used to visualise the flow of work. Typically a Kanban board shows how work moves from left to right, each column represents a stage within the value stream.

The image below is a typical view of a simplified Kanban board, where work items move from left to right. In some cases each column has a work in progress limit. This means that each column can only receive a fixed amount of work items with the aim to encourage focus, and make system constraints evident.

### Examples

#### Software development

The diagram here and the one in the Kanban Board section shows a software development workflow.[12] The boards, designed for the context in which they are used, vary considerably and may show work item types ("features" and "user stories" here), columns delineating workflow activities, explicit policies, and swimlanes (rows crossing several columns, used for grouping user stories by feature here). The aim is to make the general workflow and the progress of individual items clear to participants and stakeholders.

#### Other uses

Although it is usually used for software development and software teams, the kanban method has been applied to other aspects of knowledge work.[13] Business functions which have used kanban include:

- Human resources[14] and recruitment[15]
- Marketing
- Organizational strategy and executive leadership[16]

### See also

- Lean software development
- List of software development philosophies

### References

A minimum viable product (MVP) is a product with just enough features to satisfy early customers, and to provide feedback for future product development.[1][2]

Gathering insights from an MVP is often less expensive than developing a product with more features, which increases costs and risk if the product fails, for example, due to incorrect assumptions. The term was coined and defined in 2001 by Frank Robinson,[3] and then popularized by Steve Blank and Eric Ries.[4][5][6][7] It may also involve carrying out market analysis beforehand.

## Description

A minimum viable product has just enough core features to effectively deploy the product, and no more. Developers typically deploy the product to a subset of possible customers—such as early adopters thought to be more forgiving, more likely to give feedback, and able to grasp a product vision from an early prototype or marketing information. This strategy targets avoiding building products that customers do not want and seeks to maximize information about the customer per amount of money spent.

"The minimum viable product is that version of a new product a team uses to collect the maximum amount of validated learning about customers with the least effort."[2] The definition's use of the words maximum and minimum means it is not formulaic. It requires judgement to figure out, for any given context, what MVP makes sense. Due to this vagueness, the term MVP is commonly used, either deliberately or unwittingly, to refer to a much broader notion ranging from a rather prototype-like product to a fully-fledged and marketable product.[8]

An MVP can be part of a strategy and process directed toward making and selling a product to customers.[9] It is a core artifact in an iterative process of idea generation, prototyping, presentation, data collection, analysis and learning. One seeks to minimize the total time spent on an iteration. The process is iterated until a desirable product-market fit is
obtained, or until the product is deemed non-viable.

Steve Blank typically refers to minimum viable product as minimum feature set.[10][11]

**Purposes**

- Be able to test a product hypothesis with minimal resources
- Accelerate learning
- Reduce wasted engineering hours
- Get the product to early customers as soon as possible
- Base for other products
- To establish a builder's abilities in crafting the product required
- Brand building very quickly

**Testing**

- Results from a minimum viable product test aim to indicate if the product should be built to begin with. Testing evaluates if the initial problem or goal is solved in a manner that makes it reasonable to move forward.

**Notable quotes**

- Steve Blank: "You’re selling the vision and delivering the minimum feature set to visionaries, not everyone."[10]

**Marketing**

Releasing and assessing the impact of a minimum viable product is a market testing strategy that is used to screen product ideas soon after their generation. In software development, the release is facilitated by rapid application development tools and languages common to web application development.

The MVP differs from the conventional market testing strategy of investing time and money early to implement a product before testing it in the market. The MVP is intended to ensure that the market *wants the product* before large time and monetary investments are made. The MVP differs from the open source software methodology of *release early, release often* that listens to users, letting them define the features and future of the product. The MVP starts with a product vision, which is maintained throughout the product life cycle, although it is adapted based on the explicit and implicit (indirect measures) feedback from potential future customers of the product.[2]

The MVP is a strategy that may be used as a part of Blank's customer development methodology that focuses on continual product iteration and refinement based on customer feedback. Additionally, the presentation of non-existing products and features may be refined using web-based statistical hypothesis testing, such as A/B testing.

In software development, the general method of *deploy first, code later* is akin to the agile program code testing methodology called test-driven development where unit tests are written first and fail until the code is written.

**Business Model Canvas**

The Business Model Canvas is used to map in the major components and activities for a company starting out. The minimum viable product can be designed by using selected components of the Business Model Canvas.[12]

- Customers
Andrea Contigiani said that his PhD research showed that early release of an MVP may hurt a company more than help when companies risk imitation by a competitor and have not established other barriers to imitation. He also said that negative feedback on a MVP can negatively affect a company's reputation.

Concepts from minimum viable product are applied in other aspects of startups and organizations.

**Minimum viable brand (MVB)**

Using a minimum viable brand (MVB) concept can ensure brand hypotheses are grounded in strategic intent and market insights.

**Minimum viable co-founder**

Finding other people to create a minimum viable product is a common challenge for new companies and startups. The concept of minimum viable co-founder is based on looking for a co-founder with the following attributes:

- Trust
- Exceptional at building or selling
- Company commitment
- Personally likeable
- Productivity
- Reasonable
- Rational
- Realistic

**Minimum viable team**

Founders with an early-stage company are faced with the challenge of building a team with minimal people and cost. The process starts by listing out basic functions of a particular company (e.g., engineer, operations, finance) and then stripping down to the abstract job activities and skills that the company must have to operate.

**Criticism**

Many developers of mobile and digital products are now criticizing the MVP because customers can easily switch between competing products through platforms (e.g. app stores). Products that do not offer the expected minimum standard of quality are inferior to competitors that enter the market with a higher standard. The criticism of the MVP approach has led to several new approaches, e.g. the Minimum Viable Experiment MVE or the Minimum Awesome Product MAP.

**See also**

- Lean startup
References

7. Holiday, Ryan The single worst marketing decision you can make (https://thenextweb.com/entrepreneur/2015/04/01/the-single-worst-marketing-decision-you-can-make/) The Next Web. 1 April 2015

MoSCoW method

The MoSCoW method is a prioritization technique used in management, business analysis, project management, and software development to reach a common understanding with stakeholders on the importance they place on the delivery of each requirement; it is also known as MoSCoW prioritization or MoSCoW analysis.

The term MoSCoW itself is an acronym derived from the first letter of each of four prioritization categories (Must have, Should have, Could have, and Won’t have), with the interstitial Os added to make the word pronounceable. While the Os are usually in lower-case to indicate that they do not stand for anything, the all-capitals MOSCOW is also used.

Contents

Background

Prioritization of requirements

Variants

Use in new product development

Criticism

References

External links

Background

This prioritization method was developed by Dai Clegg[1] and first used extensively with the agile project delivery framework Dynamic Systems Development Method (DSDM).[2]

MoSCoW is often used with timeboxing, where a deadline is fixed so that the focus must be on the most important requirements, and as such is a technique commonly used in agile software development approaches such as Scrum, rapid application development (RAD), and DSDM.

Prioritization of requirements

All requirements are important, but they are prioritized to deliver the greatest and most immediate business benefits early. Developers will initially try to deliver all the Must have, Should have and Could have requirements but the Should and Could requirements will be the first to be removed if the delivery timescale looks threatened.

The plain English meaning of the prioritization categories has value in getting customers to better understand the impact of setting a priority, compared to alternatives like High, Medium and Low.

The categories are typically understood as:[3]

Must have

Requirements labeled as Must have are critical to the current delivery timebox in order for it to be a success. If even one Must have requirement is not included, the project delivery should be considered a failure (note: requirements can be downgraded from Must have, by agreement
with all relevant stakeholders; for example, when new requirements are deemed more important). MUST can also be considered an acronym for the Minimum Usable SubseT.

**Should have**
Requirements labeled as Should have are important but not necessary for delivery in the current delivery timebox. While Should have requirements can be as important as Must have, they are often not as time-critical or there may be another way to satisfy the requirement, so that it can be held back until a future delivery timebox.

**Could have**
Requirements labeled as Could have are desirable but not necessary, and could improve user experience or customer satisfaction for little development cost. These will typically be included if time and resources permit.

**Won't have (this time)**
Requirements labeled as Won't have have been agreed by stakeholders as the least-critical, lowest-payback items, or not appropriate at that time. As a result, Won't have requirements are not planned into the schedule for the next delivery timebox. Won't have requirements are either dropped or reconsidered for inclusion in a later timebox. (Note: occasionally the term Would like to have is used; however, that usage is incorrect, as this last priority is clearly stating something is outside the scope of delivery).

**Variants**
Sometimes W is used to mean Wish (or Would), i.e. still possible but unlikely to be included (and less likely than Could). This is then distinguished from X for Excluded for items which are explicitly not included.

**Use in new product development**
In new product development, particularly those following agile software development approaches, there is always more to do than there is time or funding to permit (hence the need for prioritization).

For example, should a team have too many potential epics (i.e., high-level stories) for the next release of their product, they could use the MoSCoW method to select which epics are Must have, which Should have, and so on; the minimum viable product (or MVP) would be all those epics marked as Must have.[4] Oftentimes, a team will find that, even after identifying their MVP, they have too much work for their expected capacity. In such cases, the team could then use the MoSCoW method to select which features (or stories, if that is the subset of epics in their organisation) are Must have, Should have, and so on; the minimum marketable features (or MMF) would be all those marked as Must have.[5] If there is sufficient capacity after selecting the MVP or MMF, the team could then plan to include Should have and even Could have items too.[6]

**Criticism**
Criticism of the MoSCoW method includes:

- Lack of rationale around how to rank competing requirements: why something is must rather than should.[7][8]
- Ambiguity over timing, especially on the Won't have category: whether it is not in this release or not ever.[7]
- Potential for political focus on building new features over technical improvements (such as refactoring).[8]

**References**
External links

- RFC 2119 (Requirement Levels) (http://www.ietf.org/rfc/rfc2119.txt) This RFC defines requirement levels to be used in formal documentation. It is commonly used in contracts and other legal documentation. Noted here as the wording is similar but not necessarily the meaning.
- Buffered Moscow Rules (http://manage.techwell.com/articles/membersub/time-boxing-planning-buffered-moscow-rules) This essay proposes the use of a modified set of Moscow rules that accomplish the objectives of prioritizing deliverables and providing a degree of assurance as a function of the uncertainty of the underlying estimates.
- MoSCoW Prioritisation (https://www.dsdm.org/content/moscow-prioritisation) Steps and tips for prioritisation following the DSDM MoSCoW rules.
- The ToToTo Method (http://creativehero.es/tototo-method/) A method inspired by the MoSCoW Method of prioritization.

Planning poker

Planning poker, also called Scrum poker, is a consensus-based, gamified technique for estimating, mostly used to estimate effort or relative size of development goals in software development. In planning poker, members of the group make estimates by playing numbered cards face-down to the table, instead of speaking them aloud. The cards are revealed, and the estimates are then discussed. By hiding the figures in this way, the group can avoid the cognitive bias of anchoring, where the first number spoken aloud sets a precedent for subsequent estimates.

Planning poker is a variation of the Wideband Delphi method. It is most commonly used in agile software development, in particular in Scrum and Extreme Programming.

The method was first defined and named by James Grenning in 2002[1] and later popularized by Mike Cohn in the book Agile Estimating and Planning,[2] whose company trade marked the term [3] and a digital online tool.[4]

Contents

Process
  Rationale
  Equipment
  Procedure

Benefits

See also

References

Process

Rationale

The reason to use planning poker is to avoid the influence of the other participants. If a number is spoken, it can sound like a suggestion and influence the other participants' sizing. Planning poker should force people to think independently and propose their numbers simultaneously. This is accomplished by requiring that all participants show their card at the same time.

Equipment
Planning poker is based on a list of features to be delivered, several copies of a deck of cards and optionally, an egg timer that can be used to limit time spent in discussion of each item.

The feature list, often a list of user stories, describes some software that needs to be developed.

The cards in the deck have numbers on them. A typical deck has cards showing the Fibonacci sequence including a zero: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89; other decks use similar progressions with a fixed ratio between each value such as 1, 2, 4, 8, etc.

The reason for using the Fibonacci sequence instead of simply doubling each subsequent value is because estimating a task as exactly double the effort as another task is misleadingly precise. A task which is about twice as much effort as a 5, has to be evaluated as either a bit less than double (8) or a bit more than double (13).

Several commercially available decks use the sequence: 0, ½, 1, 2, 3, 5, 8, 13, 20, 40, 100, and optionally a ? (unsure), an infinity symbol (this task cannot be completed) and a coffee cup (I need a break, and I will make the rest of the team tea). The reason for not exactly following the Fibonacci sequence after 13 is because someone once said to Mike Cohn "You must be very certain to have estimated that task as 21 instead of 20." Using numbers with only a single digit of precision (except for 13) indicates the uncertainty in the estimation. Some organizations use standard playing cards of Ace, 2, 3, 5, 8 and king. Where king means: "this item is too big or too complicated to estimate". "Throwing a king" ends discussion of the item for the current sprint.

Smartphones allow developers to use mobile apps instead of physical card decks. When teams are not in the same geographical locations, collaborative software can be used as replacement for physical cards.

**Procedure**

At the estimation meeting, each estimator is given one deck of the cards. All decks have identical sets of cards in them.

The meeting proceeds as follows:

- A Moderator, who will not play, chairs the meeting.
- The Product Owner provides a short overview of one user story to be estimated. The team is given an opportunity to ask questions and discuss to clarify assumptions and risks. A summary of the discussion is recorded, e.g. by the Moderator.
- Each individual lays a card face down representing their estimate for the story. Units used vary - they can be days duration, ideal days or story points. During discussion, numbers must not be mentioned at all in relation to feature size to avoid anchoring.
- Everyone calls their cards simultaneously by turning them over.
- People with high estimates and low estimates are given a soap box to offer their justification for their estimate and then discussion continues.
- Repeat the estimation process until a consensus is reached. The developer who was likely to own the deliverable has a large portion of the "consensus vote", although the Moderator can negotiate the consensus.
- To ensure that discussion is structured; the Moderator or the Product Owner may at any point turn over the egg timer and when it runs out all discussion must cease and another round of poker is played. The structure in the conversation is re-introduced by the soap boxes.

The cards are numbered as they are to account for the fact that the longer an estimate is, the more uncertainty it contains. Thus, if a developer wants to play a 6 he is forced to reconsider and either work through that some of the perceived uncertainty does not exist and play a 5, or accept a conservative estimate accounting for the uncertainty and play an 8.

**Benefits**
A study by Moløkken-Østvold and Haugen\cite{5} reported that planning poker provided accurate estimates of programming task completion time, although estimates by any individual developer who entered a task into the task tracker was just as accurate. Tasks discussed during planning poker rounds took longer to complete than those not discussed and included more code deletions, suggesting that planning poker caused more attention to code quality. Planning poker was considered by the study participants to be effective at facilitating team coordination and discussion of implementation strategies.

See also

- Comparison of Scrum software, which generally has support for planning poker, either included or as an optional add-on.

References


This page was last edited on 7 March 2019, at 04:35 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.
Scrum (software development)

Scrum is an agile framework for managing knowledge work, with an emphasis on software development. It is designed for teams of three to nine members, who break their work into actions that can be completed within timeboxed iterations, called sprints, no longer than one month and most commonly two weeks, then track progress and re-plan in 15-minute time-boxed stand-up meetings, called daily scrums.[1][2]

Approaches to coordinating the work of multiple scrum teams in larger organizations include Large-scale Scrum (LeSS), Scaled agile framework (SAFe), scrum of scrums, and Scrum@Scale, the Nexus, among others.
Key ideas

Scrum is a lightweight, iterative and incremental framework for managing product development.[3][4] It defines "a flexible, holistic product development strategy where a development team works as a unit to reach a common goal",[5] challenges assumptions of the "traditional, sequential approach"[5] to product development, and enables teams to self-organize by encouraging physical co-location or close online collaboration of all team members, as well as daily face-to-face communication among all team members and disciplines involved.

A key principle of Scrum is the dual recognition that customers will change their minds about what they want or need (often called requirements volatility[6]) and that there will be unpredictable challenges—for which a predictive or planned approach is not suited. As such, Scrum adopts an evidence-based empirical approach—accepting that the problem cannot be fully understood or defined up front, and instead focusing on how to maximize the team's ability to deliver quickly, to respond to emerging requirements, and to adapt to evolving technologies and changes in market conditions.

Many of the terms used in Scrum (e.g., scrum master) are typically written with leading capitals (e.g., Scrum Master). To maintain an encyclopedic tone, however, this article uses normal sentence case for these terms—unless they are recognized marks (such as Certified Scrum Master). This is occasionally seen written in all-capitals, as SCRUM.[7] The word is not an acronym, so this is not correct; however, it likely arose due to an early paper by Ken Schwaber which capitalized SCRUM in its title.[1][8]

While the trademark on the term Scrum itself has been allowed to lapse, so that it is deemed as owned by the wider community rather than an individual,[9] the leading capital is retained—except when used with other words (as in daily scrum or scrum team).

History

Hirotaka Takeuchi and Ikujiro Nonaka introduced the term scrum in the context of product development in their 1986 Harvard Business Review article, "The New New Product Development Game".[5] Takeuchi and Nonaka later argued in The Knowledge Creating Company[10] that it is a form of "organizational knowledge creation, [...] especially good at bringing about innovation continuously, incrementally and spirally".

The authors described a new approach to commercial product development that would increase speed and flexibility, based on case studies from manufacturing firms in the automotive, photocopier and printer industries.[5] They called this the holistic or rugby approach, as the whole process is performed by one cross-functional team across multiple overlapping phases, where the team "tries to go the distance as a unit, passing the ball back and forth".[5] (In rugby football, a scrum is used to restart play, as the forwards of each team interlock with their heads down and attempt to gain possession of the ball.[11])

In the early 1990s, Ken Schwaber used what would become Scrum at his company, Advanced Development Methods; while Jeff Sutherland, John Scumniotales and Jeff McKenna, developed a similar approach at Easel Corporation, referring to it using the single word Scrum.[12]

In 1995, Sutherland and Schwaber jointly presented a paper describing the Scrum framework at the Business Object Design and Implementation Workshop held as part of Object-Oriented Programming, Systems, Languages & Applications '95 (OOPSLA '95) in Austin, Texas.[13] Over the following years, Schwaber and Sutherland collaborated to combine this material—with their experience and evolving good practice—to develop what became known as Scrum.[14]
In 2001, Schwaber worked with Mike Beedle to describe the method in the book, *Agile Software Development with Scrum*. Scrum's approach to planning and managing product development involves bringing decision-making authority to the level of operation properties and certainties.

In 2002, Schwaber with others founded the Scrum Alliance and set up the Certified Scrum accreditation series. Schwaber left the Scrum Alliance in late 2009 and founded Scrum.org which oversees the parallel Professional Scrum accreditation series.

Since 2009, a public document called *The Scrum Guide* has officially defined Scrum. It has been revised 5 times, with the current version being November 2017.

In 2018, Schwaber and the Scrum.org community, along with leaders of the Kanban community, published *The Kanban Guide for Scrum Teams*. 

## Roles

There are three roles in the Scrum framework. These are ideally co-located to ensure optimal communication among team members. Together these three roles form the Scrum team. While many organizations have other roles involved with defining and delivering the product, Scrum defines only these three.

### Product owner

The product owner represents the product's stakeholders and the voice of the customer, is responsible for the product backlog and accountable for maximizing the value that the team delivers. The product owner defines the product in customer-centric terms (typically user stories), adds them to the product backlog, and prioritizes them based on importance and dependencies. A scrum team should have only one product owner (although a product owner could support more than one team). This role should not be combined with that of the scrum master. The product owner should focus on the business side of product development and spend the majority of their time liaising with stakeholders and the engineers which are part of their Team. The product owner should not dictate how the team reaches a technical solution, but rather will seek consensus among the team members. This role is crucial and requires a deep understanding of both sides: the business and the engineers (developers) in the Scrum Team. Therefore a good product owner should be able to communicate what the business need, ask why they need it (because there may be better ways to achieve that), and convey the message to all stakeholders including the delivery Team using a technical language, as required. The role is crucial because the product owner needs to be technical enough to understand the architecture of the solution in order to be credible by the Team.

Communication is a core responsibility of the product owner. The ability to convey priorities and empathize with team members and stakeholders is vital to steer product development in the right direction. The product owner role bridges the communication gap between the team and its stakeholders, serving as a proxy for stakeholders to the team and as a team representative to the overall stakeholder community.

As the face of the team to the stakeholders, the following are some of the communication tasks of the product owner to the stakeholders:

- organize further meetings to demonstrate the solution to key stakeholders who were not present at a sprint review;
- defines and announces releases;
- communicates delivery and team status;
- share progress during governance meetings
- share RAID (Risks, Assumptions, Issues and Dependencies) with stakeholders
negotiates priorities, scope, funding, and schedule;
ensures that the product backlog is visible, transparent and clear.

Empathy is a key attribute for a product owner to have—the ability to put one's self in another's shoes. A product owner converses with different stakeholders, who have a variety of backgrounds, job roles, and objectives. A product owner must be able to see from these different points of view. To be effective, it is wise for a product owner to know the level of detail the audience needs. The development team needs thorough feedback and specifications so they can build a product up to expectation, while an executive sponsor may just need summaries of progress. Providing more information than necessary may lose stakeholder interest and waste time. A direct means of communication is the most preferred by seasoned agile product owners.[20]

A product owner's ability to communicate effectively is also enhanced by being skilled in techniques that identify stakeholder needs, negotiate priorities between stakeholder interests, and collaborate with developers to ensure effective implementation of requirements.

Development team

The development team is responsible for delivering potentially releasable product increments every sprint. The sprint goal is a short description of what the team is supposed to deliver at the end of the sprint, and is important for both the Team and the stakeholders.

The team has from three to nine members who carry out all tasks required to build the product increments (analysis, design, development, testing, technical writing, etc.).[^19] Although there will be several disciplines represented in the team, its members are referred to generically as developers. To avoid potential confusion that this only refers to programmers, some organizations call this a delivery team and its members just team members.

The development team in Scrum is self-organizing, and any interaction outside the team is managed by the scrum master (who will protect the team), or the product owner (who has regular interactions with all stakeholders).

Scrum master

Scrum is facilitated by a scrum master, who is accountable for removing impediments to the ability of the team to deliver the product goals and deliverables. The scrum master is not a traditional team lead or project manager but acts as a buffer between the team and any distracting influences. The scrum master ensures that the Scrum framework is followed. The scrum master helps to ensure the team follows the agreed processes in the Scrum framework, often facilitates key sessions, and encourages the team to improve. The role has also been referred to as a team facilitator[^25] or servant-leader to reinforce these dual perspectives.

The core responsibilities of a scrum master include (but are not limited to):[^26]

- Helping the product owner maintain the product backlog in a way that ensures the needed work is well understood so the team can continually make forward progress
- Helping the team to determine the definition of done for the product, with input from key stakeholders
- Coaching the team, within the Scrum principles, in order to deliver high-quality features for its product
- Promoting self-organization within the team
- Helping the scrum team to avoid or remove impediments to its progress, whether internal or external to the team
- Facilitating team events to ensure regular progress
- Educating key stakeholders on Agile and Scrum principles
- Coaching the development team in self-organization and cross-functionality
One of the ways the scrum master role differs from a project manager is that the latter may have people management responsibilities and the scrum master does not. Scrum does not formally recognise the role of project manager, as traditional command and control tendencies would cause difficulties.[27]

**Workflow**

**Sprint**

A sprint (or iteration) is the basic unit of development in Scrum. The sprint is a timeboxed effort; that is, it is restricted to a specific duration.[28] The duration is fixed in advance for each sprint and is normally between one week and one month, with two weeks being the most common.[1]

Each sprint starts with a sprint planning event that aims to define a sprint backlog, identify the work for the sprint, and make an estimated forecast for the sprint goal. Each sprint ends with a sprint review and sprint retrospective,[12] that reviews progress to show to stakeholders and identify lessons and improvements for the next sprints.

Scrum emphasizes working product at the end of the sprint that is really done. In the case of software, this likely includes that the software has been fully integrated, tested and documented, and is potentially releasable.[27]

**Sprint planning**

At the beginning of a sprint, the scrum team holds a sprint planning event[28] to:

- Mutually discuss and agree on the scope of work that is intended to be done during that sprint
- Select product backlog items that can be completed in one sprint
- Prepare a sprint backlog that includes the work needed to complete the selected product backlog items
- The recommended duration is four hours for a two-week sprint (pro-rata for other sprint durations) [14]

  - During the first half, the whole scrum team (development team, scrum master, and product owner) selects the product backlog items they believe could be completed in that sprint
  - During the second half, the development team identifies the detailed work (tasks) required to complete those product backlog items; resulting in a confirmed sprint backlog
  - As the detailed work is elaborated, some product backlog items may be split or put back into the product backlog if the team no longer believes they can complete the required work in a single sprint
  - Once the development team has prepared their sprint backlog, they forecast (usually by voting) which tasks will be delivered within the sprint.

**Daily Scrum**

Each day during a sprint, the team holds a daily scrum (or stand-up) with specific guidelines:

- All members of the development team come prepared. The daily scrum:
  - starts precisely on time even if some development team members are missing
  - should happen at the same time and place every day
is limited (timeboxed) to fifteen minutes

Anyone is welcome, though only development team members should contribute.

During the daily scrum, each team member typically answers three questions:

- What did I complete yesterday that contributed to the team meeting our sprint goal?
- What do I plan to complete today to contribute to the team meeting our sprint goal?
- Do I see any impediment that could prevent me or the team from meeting our sprint goal?

Any impediment (e.g., stumbling block, risk, issue, delayed dependency, assumption proved unfounded) identified in the daily scrum should be captured by the scrum master and displayed on the team's scrum board or on a shared risk board, with an agreed person designated to working toward a resolution (outside of the daily scrum). No detailed discussions should happen during the daily scrum.

Sprint review

At the end of a sprint, the team holds two events: the sprint review and the sprint retrospective.

At the sprint review, the team:

- reviews the work that was completed and the planned work that was not completed
- presents the completed work to the stakeholders (a.k.a. the demo)
- collaborates with the stakeholders on what to work on next

Guidelines for sprint reviews:

- Incomplete work cannot be demonstrated.
- The recommended duration is two hours for a two-week sprint (proportional for other sprint durations).\(^{[14]}\)

Sprint retrospective

At the sprint retrospective, the team:

- Reflects on the past sprint
- Identifies and agrees on continuous process improvement actions

Guidelines for sprint retrospectives:

- Three main questions are asked in the sprint retrospective: What went well during the sprint? What did not go well? What could be improved for better productivity in the next sprint?
- The recommended duration is one-and-a-half hours for a two-week sprint (proportional for other sprint duration(s))
- This event is facilitated by the scrum master

Extensions

The following activities are commonly done, although not considered as a core part of Scrum:

**Backlog refinement**
Backlog refinement (formerly called grooming) is the ongoing process of reviewing product backlog items and checking that they are appropriately prepared and ordered in a way that makes them clear and executable for teams once they enter sprints via the sprint planning activity. Product backlog items may be broken into multiple smaller ones. Acceptance criteria may be clarified. Dependencies may be identified and investigated.

Although not originally a core Scrum practice, backlog refinement has been added to the Scrum Guide and adopted as a way of managing the quality of product backlog items entering a sprint, with a recommended investment of up to 10% of a team’s sprint capacity.\[14\][30]

The backlog can also include technical debt (also known as design debt or code debt). This is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer.

**Cancelling a sprint**

The product owner can cancel a sprint if necessary.\[14\] The product owner may do so with input from the team, scrum master or management. For instance, management may wish the product owner to cancel a sprint if external circumstances negate the value of the sprint goal. If a sprint is abnormally terminated, the next step is to conduct a new sprint planning, where the reason for the termination is reviewed.

**Artifacts**

**Product backlog**

The product backlog is a model of work to be done\[31\] and contains an ordered list of product requirements that a scrum team maintains for a product. The format of product backlog items varies, common formats include user stories, use cases, or any other requirements format the team finds useful.\[27\] These will define features, bug fixes, non-functional requirements, etc.—whatever must be done to successfully deliver a viable product. The product owner prioritizes product backlog items (PBIs) based on considerations such as risk, business value, dependencies, size, and date needed.

The product backlog is what will be delivered, ordered into the sequence in which it should be delivered. It is visible to everyone but may only be changed with the consent of the product owner, who is ultimately responsible for ordering product backlog items for the development team to choose.

The product backlog contains the product owner’s assessment of business value and the development team’s assessment of development effort, which are often, but not always, stated in story points using the rounded Fibonacci scale. These estimates help the product owner to gauge the timeline and may influence the ordering of product backlog items; for example, if two features have the same business value, the product owner may schedule earlier delivery of the one with the lower development effort (because the return on investment is higher) or the one with higher development effort (because it is more complex or riskier, and they want to retire that risk earlier).\[32\]

The product backlog and the business value of each product backlog item is the responsibility of the product owner. The effort to deliver each item is estimated by the development team in story points, or time. By estimating in story points the product owner decouple each item from the developer, and this can be useful especially in dynamic teams where developers are often assigned to other projects after sprint delivery. For instance if a user story is 5 (using Fibonacci sequence) it remains 5 regardless of how many developers are working on it. Story points define the effort in a time-box, so they do not change with time. For instance in one hour we can walk, run, or climb, but the effort is clearly different. Also the gap between each number in Fibonacci (or other sequences), is suggesting the team to deliver accurate estimates. For
example 1, 2 or 3 have similar efforts, 1 being trivial, but if the team estimate an 8 for 13, the impact on both delivery and budget can be significant. The value of using story points is that we can reuse them by comparing similar work from previous sprints, but we need to remember that estimates are relative to the team. For instance a 5 for one team, could be a 2 for another having senior developers and higher skills.

Every team should have a product owner, although in many instances they will work with more than one team. The product owner is responsible for maximizing the value of the product. The product owner gathers input and takes feedback from, and is lobbied by, many people, but ultimately makes the call on what gets built.

The product backlog:

- Captures requests to modify a product—including new features, replacing old features, removing features, and fixing issues
- Ensures the development team has work that maximizes business benefit to the product owner

Typically, the product owner and the scrum team come together and write down everything that must be prioritized, and this becomes content for the first sprint—which is a block of time meant for focused work on selected items that can be accommodated within a timeframe. The product backlog can evolve as new information surfaces about the product and about its customers, and so later sprints may address new work.

**Management**

A product backlog, in its simplest form, is merely a list of items to work on. Having well-established rules about how work is added, removed and ordered helps the whole team make better decisions about how to change the product.

The product owner prioritizes product backlog items based on which are needed soonest. The team then chooses which items they can complete in the coming sprint. On the scrum board, the team moves items from the product backlog to the sprint backlog, which is the list of items they will build. Conceptually, it is ideal for the team to only select what they think they can accomplish from the top of the list, but it is not unusual to see in practice that teams are able to take lower-priority items from the list along with the top ones selected. This normally happens because there is time left within the sprint to accommodate more work. Items at the top of the backlog, the items to work on first, should be broken down into stories that are suitable for the development team to work on. The further down the backlog goes, the less refined the items should be. As Schwaber and Beedle put it "The lower the priority, the less detail until you can barely make out the backlog item."

As the team works through the backlog, it must be assumed that change happens outside their environment—the team can learn about new market opportunities to take advantage of, competitor threats that arise, and feedback from customers that can change the way the product was meant to work. All of these new ideas tend to trigger the team to adapt the backlog to incorporate new knowledge. This is part of the fundamental mindset of an agile team. The world changes, the backlog is never finished.

**Sprint backlog**

The sprint backlog is the list of work the development team must address during the next sprint. The list is derived by the scrum team progressively selecting product backlog items in priority order from the top of the product backlog until they feel they have enough work to fill the sprint. The development team should keep in mind its past performance assessing its capacity for the new-sprint, and use this as a guideline of how much 'effort' they can complete.
The product backlog items may be broken down into tasks by the development team. Tasks on the sprint backlog are never assigned (or pushed) to team members by someone else; rather team members sign up for (or pull) tasks as needed according to the backlog priority and their own skills and capacity. This promotes self-organization of the development team and developer buy-in.

The sprint backlog is the property of the development team, and all included estimates are provided by the development team. Often an accompanying task board is used to see and change the state of the tasks of the current sprint, like to do, in progress and done.

Once a sprint backlog is committed, no additional work can be added to the sprint backlog except by the team. Once a sprint has been delivered, the product backlog is analyzed and reprioritized if necessary, and the next set of functionality is selected for the next sprint.

**Product increment**

The potentially releasable increment is the sum of all the product backlog items completed during a sprint, integrated with the work of all previous sprints. At the end of a sprint, the increment must be complete, according to the scrum team’s definition of "done" (DoD), fully functioning, and in a usable condition regardless of whether the product owner decides to actually release it.

**Extensions**

The following artifacts are commonly used, although not considered by all as a core part of Scrum:

**Sprint burn-down chart**

The sprint burn-down chart is a publicly displayed chart showing remaining work in the sprint backlog. Updated every day, it gives a simple view of the sprint progress. It also provides quick visualizations for reference. The horizontal axis of the sprint burn-down chart shows the days in a sprint, while the vertical axis shows the amount of work remaining each day (typically representing the estimate of hours of work remaining).

During sprint planning, the ideal burndown chart is plotted. Then, during the sprint, each member picks up tasks from the sprint backlog and works on them. At the end of the day, they update the remaining hours for tasks to be completed. In such a way, the actual burndown chart is updated day by day.

It should not be confused with an earned value chart.

**Release burn-up chart**

The release burn-up chart is a way for the team to provide visibility and track progress toward a release. Updated at the end of each sprint, it shows progress toward delivering a forecast scope. The horizontal axis of the release burn-up chart shows the sprints in a release, while the vertical axis shows the amount of work completed at the end of each sprint (typically representing cumulative story points of work completed). Progress is plotted as a line that grows up to meet a
horizontal line that represents the forecast scope; often shown with a forecast, based on progress to date, that indicates how much scope might be completed by a given release date or how many sprints it will take to complete the given scope.

The release burn-up chart makes it easy to see how much work has been completed, how much work has been added or removed (if the horizontal scope line moves), and how much work is left to be done.

**Definition of done (DoD)**

The exit-criteria to determine whether a product backlog item is complete. In many cases, the DoD requires that all regression tests be successful. The definition of done may vary from one scrum team to another but must be consistent within one team.[36]

**Velocity**

The total effort a team is capable of in a sprint. The number is derived by evaluating the work (typically in user story points) completed in the last sprint. The collection of historical velocity data is a guideline for assisting the team in understanding how much work they can likely achieve in a future sprint.

**Spike**

A time-boxed period used to research a concept or create a simple prototype. Spikes can either be planned to take place in between sprints or, for larger teams, a spike might be accepted as one of many sprint delivery objectives. Spikes are often introduced before the delivery of large or complex product backlog items in order to secure budget, expand knowledge, or produce a proof of concept. The duration and objective(s) of a spike is agreed between product owner and development team before the start. Unlike sprint commitments, spikes may or may not deliver tangible, shippable, valuable functionality. For example, the objective of a spike might be to successfully reach a decision on a course of action. The spike is over when the time is up, not necessarily when the objective has been delivered.[37]

**Tracer bullet**

Also called a drone spike, a tracer bullet is a spike with the current architecture, current technology set, current set of best practices that result in production quality code. It might just be a very narrow implementation of the functionality but is not throwaway code. It is of production quality, and the rest of the iterations can build on this code. The name has military origins as ammunition that makes the path of the bullet visible, allowing for corrections. Often these implementations are a 'quick shot' through all layers of an application, such as connecting a single form's input field to the back-end, to prove the layers connect as expected.[38]

**Limitations**

Scrum works less well in the following circumstances:[39][40]

- **Teams whose members are geographically dispersed or part-time:** In Scrum, developers should have close and ongoing interaction, ideally working together in the same space most of the time. While recent improvements in technology have reduced the impact of these barriers (e.g., being able to collaborate on a digital whiteboard), the Agile manifesto asserts that the best communication is face to face.[41]
- **Teams whose members have very specialized skills:** In Scrum, developers should have T-shaped skills, allowing them to work on tasks outside of their specialization. This can be encouraged by good Scrum leadership. While team
members with very specific skills can and do contribute well, they should be encouraged to learn more about and collaborate with other disciplines.

- **Products with many external dependencies**: In Scrum, dividing product development into short sprints requires careful planning; external dependencies, such as user acceptance testing or coordination with other teams, can lead to delays and the failure of individual sprints.

- **Products that are mature or legacy or with regulated quality control**: In Scrum, product increments should be fully developed and tested in a single sprint; products that need large amounts of regression testing or safety testing (e.g., medical devices or vehicle control) for each release are less suited to short sprints than to longer waterfall releases.

From a business perspective, Scrum has many virtues, one of which is that it is designed to yield the best business solutions. However, the efficiency by which it does so in any given organization can vary widely and is largely dependent on the ability of the organization to adhere to the implementation guidelines. Every company has its own distinct organizational structure, culture, and set of business practices, and some are more naturally amenable to this methodology than others.

## Tools for implementation

Like other agile methods, effective adoption of Scrum can be supported through a wide range of tools.

Many companies use universal tools, such as spreadsheets to build and maintain artifacts such as the sprint backlog. There are also open-source and proprietary software packages for Scrum—which are either dedicated to product development using the Scrum framework or support multiple product development approaches including Scrum.

Other organizations implement Scrum without software tools and maintain their artifacts in hard-copy forms such as paper, whiteboards, and sticky notes.[42]

## Scrum values

Scrum is a feedback-driven empirical approach which is, like all empirical process control, underpinned by the three pillars of transparency, inspection, and adaptation. All work within the Scrum framework should be visible to those responsible for the outcome: the process, the workflow, progress, etc. In order to make these things visible, scrum teams need to frequently inspect the product being developed and how well the team is working. With frequent inspection, the team can spot when their work deviates outside of acceptable limits and adapt their process or the product under development.[19]

These three pillars require trust and openness in the team, which the following five values of Scrum enable:[14]

1. **Commitment**: Team members individually commit to achieving their team goals, each and every sprint.
2. **Courage**: Team members know they have the courage to work through conflict and challenges together so that they can do the right thing.
3. **Focus**: Team members focus exclusively on their team goals and the sprint backlog; there should be no work done other than through their backlog.
4. **Openness**: Team members and their stakeholders agree to be transparent about their work and any challenges they face.
5. **Respect**: Team members respect each other to be technically capable and to work with good intent.

## Adaptations

The hybridization of Scrum with other software development methodologies is common as Scrum does not cover the whole product development lifecycle; therefore, organizations find the need to add in additional processes to create a more comprehensive implementation. For example, at the start of product development, organizations commonly add process
guidance on the business case, requirements gathering and prioritization, initial high-level design, and budget and schedule forecasting.[43]

Various authors and communities of people who use Scrum have also suggested more detailed techniques for how to apply or adapt Scrum to particular problems or organizations. Many refer to these methodological techniques as 'patterns' - by analogy with design patterns in architecture and software.[44][45] Such patterns have extended Scrum outside of the software development domain into Manufacturing,[46] Finance and Human Resources.

**Scrumban**

Scrumban is a software production model based on Scrum and Kanban. Scrumban is especially suited for product maintenance with frequent and unexpected work items, such as production defects or programming errors. In such cases the time-limited sprints of the Scrum framework may be perceived to be of less benefit, although Scrum's daily events and other practices can still be applied, depending on the team and the situation at hand. Visualization of the work stages and limitations for simultaneous unfinished work and defects are familiar from the Kanban model. Using these methods, the team's workflow is directed in a way that allows for minimum completion time for each work item or programming error, and on the other hand ensures each team member is constantly employed.[47]

To illustrate each stage of work, teams working in the same space often use post-it notes or a large whiteboard.[48] In the case of decentralized teams, stage-illustration software such as Assembla, JIRA or Agilo.

The major differences between Scrum and Kanban is that in Scrum work is divided into sprints that last a fixed amount of time, whereas in Kanban the flow of work is continuous. This is visible in work stage tables, which in Scrum are emptied after each sprint, whereas in Kanban all tasks are marked on the same table. Scrum focuses on teams with multifaceted know-how, whereas Kanban makes specialized, functional teams possible.[47]

**Scrum of scrums**

The scrum of scrums is a technique to operate Scrum at scale, for multiple teams working on the same product, allowing them to discuss progress on their interdependencies, focusing on how to coordinate delivering software,[49] especially on areas of overlap and integration. Depending on the cadence (timing) of the scrum of scrums, the relevant daily scrum for each scrum team ends by designating one member as an ambassador to participate in the scrum of scrums with ambassadors from other teams. Depending on the context, the ambassadors may be technical contributors or each team's scrum master.[49]

Rather than simply a progress update, the scrum of scrums should focus on how teams are collectively working to resolve, mitigate, or accept any risks, impediments, dependencies, and assumptions (RIDAs) that have been identified. The scrum of scrums tracks these RIDAs via a backlog of its own, such as a risk board (sometimes known as a ROAM board after the initials of resolved, owned, accepted, and mitigated),[50] which typically leads to greater coordination and collaboration between teams.[49]

This should run similar to a daily scrum, with each ambassador answering the following four questions:[51]

- What risks, impediments, dependencies, or assumptions has your team resolved since we last met?
- What risks, impediments, dependencies, or assumptions will your team resolve before we meet again?
- Are there any new risks, impediments, dependencies, or assumptions slowing your team down or getting in their way?
- Are you about to introduce a new risk, impediment, dependency, or assumption that will get in another team's way?

As Jeff Sutherland commented,[49]
Since I originally defined the Scrum of Scrums (Ken Schwaber was at IDX working with me), I can definitively say the Scrum of Scrums is not a "meta Scrum". The Scrum of Scrums as I have used it is responsible for delivering the working software of all teams to the Definition of Done at the end of the sprint, or for releases during the sprint. PatientKeeper delivered to production four times per Sprint. Ancestry.com delivers to production 220 times per two-week Sprint. Hubspot delivers live software 100-300 times a day. The Scrum of Scrums Master is held accountable for making this work. So the Scrum of Scrums is an operational delivery mechanism.

Large-scale Scrum

Large-scale Scrum (LeSS) is a product development framework that extends Scrum with scaling rules and guidelines without losing the original purposes of Scrum.

There are two levels to the framework: the first LeSS level is designed for up to 8 teams; the second level, known as "LeSS Huge", introduces additional scaling elements for development with up to hundreds of developers. "Scaling Scrum starts with understanding and being able to adopt standard real one-team Scrum. Large-scale Scrum requires examining the purpose of single-team Scrum elements and figuring out how to reach the same purpose while staying within the constraints of the standard Scrum rules."

Bas Vodde and Craig Larman evolved the LeSS framework from their experiences working with large-scale product development, especially in the telecoms and finance industries. It evolved by taking Scrum and trying many different experiments to discover what works. In 2013, the experiments were solidified into the LeSS framework rules. The intention of LeSS is to "descale" organization complexity, dissolving unnecessary complex organizational solutions, and solving them in simpler ways. Less roles, less management, less organizational structures.

Scrum in the classroom

Many courses in higher education are adapting the scrum framework to give students in both IT and non-IT environments new tools and better insight for dealing with project management. The scrum framework helps students grasp the concepts of the coursework in new ways, while also fostering better teamwork, better communication, breaking down tasks into smaller parts, and it helps students become more self-driven and self-organized. Scrum has been used to help students become more self-aware and it encourages self-directed learning. Many of the aforementioned skills that are gained by classroom adapted scrum are sought after by many companies and help students become more prepared for the workplace, post-graduation. It is worth noting that classroom adapted scrum is not pure scrum, as it has to be adapted to fit a 16-week period and often many members of the team must play many roles—this does however help reinforce the Agile philosophy of always being ready for change, constant adaption, and being ready for new requirements and constraints.

See also

- Disciplined agile delivery
- Lean software development
- Project management
- Unified Process
- High-performance teams
- Agile testing
References


36. Ken Schwaber, Agile Project Management with Scrum, p.55


41. Kent Beck; James Grenning; Robert C. Martin; Mike Beedle; Jim Highsmith; Steve Mellor; Arie van Bennekum; Andrew Hunt; Ken Schwaber; Alistair Cockburn; Ron Jeffries; Jeff Sutherland; Ward Cunningham; Jon Kern; Dave Thomas; Martin Fowler; Brian Marick (2001). "Principles behind the Agile Manifesto" (http://agilemanifesto.org/principles.html). Agile Alliance. Retrieved August 7, 2017.
Further reading


External links

- Agile Alliance's Scrum library (http://cf.agilealliance.org/articles/article_list.cfm?CategoryId=17)
- A Scrum Process Description (http://epf.eclipse.org/wikis/scrum/) by the Eclipse Process Framework (EPF) Project
Scrumban is an Agile management methodology describing hybrids of Scrum and Kanban and was originally designed as a way to transition from Scrum to Kanban. Today, Scrumban is a management framework that emerges when teams employ Scrum as their chosen way of working and use the Kanban Method as a lens through which to view, understand and continuously improve how they work.[1] [2]

## Contents

### History

A Framework for [R]Evolution

The Method
- Iterations
- On-demand planning
- Prioritization
- Bucket size planning
- The board
- The team
- Pull principle
- Feature freeze
- Triage

Terms

Tooling

See also

References

### History

As the Kanban method was becoming more popular, Scrumban was developed [3] as an attempt to make it easier for existing Scrum teams to begin exploring Lean and Kanban concepts.

The first article on Scrumban, which uses the spelling "Scrum-ban", describes several levels to transition from Scrum to Kanban.[1]

Fundamentally, Scrumban is a management framework that emerges when teams employ Scrum as their chosen way of working and use the Kanban Method as a lens through which to view, understand and continuously improve how they work.

Scrumban is distinct from Scrum in the way it emphasizes certain principles and practices that are substantially different from Scrum's traditional foundation. Among these are:

- recognizing the important role of organizational management (self-organization remains an objective but within the context of specific boundaries)
- allowing for specialized teams and functions
- applying explicit policies around ways of working
applying the laws of flow and queuing theory

deliberate economic prioritization[2]

Scrumban is distinct from the Kanban Method in that it:

- prescribes an underlying software development process framework (Scrum) as its core
- is organized around teams
- recognizes the value of time-boxed iterations when appropriate
- formalizes continuous improvement techniques within specific ceremonies[2]

Perhaps most importantly, the principles and practices embedded within Scrumban are not unique to the software development process. They can be easily applied in many different contexts, providing a common language and shared experience across interrelated business functions. This, in turn, enhances the kind of organizational alignment that is an essential characteristic of success.

A Framework for [R]Evolution

When Corey Ladas introduced the world to Scrumban in his seminal book of that name,[4] he defined it as a transition method for moving software development teams from Scrum to a “more evolved” software development framework. In actual practice, however, Scrumban has itself evolved to become a family of principles and practices that create complementary capabilities unique from both Scrum and the Kanban Method. These capabilities have led to three distinct manifestations:[2]

- As a framework that helps teams and organizations effectively adopt Scrum as a development methodology.
- As a framework that helps teams and organizations overcome a variety of common challenges scaling Scrum across the Enterprise.
- As a framework that helps teams and organizations develop their own set of Scrum-based processes and practices that work best for them—not to accommodate inadequacies and dysfunctions Scrum exposed, but to resolve them in a manner that was most effective for their unique environment.

The Method

In Scrumban, the teamwork is organized in small iterations and monitored with the help of a visual board, similar to Scrum and kanban boards. To illustrate each stage of work, teams working in the same space often use post-it notes or a large whiteboard. In the case of decentralized teams, visual management software such as Assembla, Targetprocess, Eylean Board, JIRA, Mingle or Agilo for Trac are often used.[1] Planning meetings are held to determine what user stories to complete in the next iteration. The user stories are then added to the board and the team completes them, the team working on a few user stories at a time as practical (the work-in-progress, or WIP, limit). To keep iterations short, WIP limits are thus used, and a planning trigger is set in place for the team to know when to plan next - when WIP falls below a predetermined level. There are no predefined roles in Scrumban; the team keeps the roles they already have.[5]

Iterations

Work iterations in Scrumban are kept short. This ensures that a team can easily adapt and change their course of action to a quickly changing environment. The length of the iteration is measured in weeks. The ideal length of an iteration depends on the work process of each team, and it is recommended not to have iterations exceeding two weeks.[6] Velocity (a measure of productivity) is often used by the team to assess issues and trends in its throughput, in order to support continuous improvement.
**On-demand planning**

The planning in Scrumban is based on demand and occurs only when the planning trigger goes off. The planning trigger is associated with the number of tasks left in the "To Do" section of the board - when it goes down to a certain number, the planning event is held. The number of tasks that should trigger a planning event is not predefined. It depends on a team’s velocity (how quickly it can finish the remaining tasks) and on the time required to plan the next iteration. The tasks planned for the next iteration are added to the "To Do" section of the board.

**Prioritization**

It is recommended to prioritize tasks during the planning event. This means the tasks are added to the board with marked priorities. It helps the team members to know which tasks should be completed first and which can be completed later. The prioritization can be done by adding numbers to the tasks or by adding an additional priority column, where the most important tasks are put at the top and the less important tasks below.

**Bucket size planning**

Bucket size planning brings the possibility of long-term planning to Scrumban. It is based on the system of three buckets that the work items need to go through before making it on the Scrumban board. The three buckets represent three different stages of the plan and are usually called 1-year, 6-month and 3-month buckets. The 1-year bucket is dedicated for long-term goals that the company has, like penetrating a new market, releasing new product, etc. When the company decides to move forward with a plan, it is moved to the 6-month bucket, where the main requirements of this plan are crystallized. When a company is ready to start implementing the plan, the requirements are moved into the 3-month bucket and divided into clear tasks to be completed by the project team. It is from this bucket that the team draws tasks during their on-demand planning meeting and starts working on the tasks.[7]

**The board**

The basic Scrumban board is composed out of three columns: To Do, Doing and Done. After the planning meeting, the tasks are added to the To Do column, when a team member is ready to work on a task, he/she moves it to the Doing column and when he/she completes it, he/she moves it to the Done column. The Scrumban board visually represents the
progress of the team. The task board columns are adapted and expanded based on the team's work progress. The most common add-ons include priority columns in the To Do section and columns like Design, Manufacturing, Testing in the Doing section.

**WIP limits** -- To ensure that the team is working effectively, Scrumban methodology states that a team member should be working on no more than one task at a time. To make sure this rule is followed Scrumban uses WIP (work in progress) limit. This limit is visualized on top of the Doing section of the board (also could be on each column of that section) and means that only that number of tasks can be in the corresponding column at one time. A WIP limit usually is equal to the number of people in the team but could be expanded based on the specifics of the team's work.

**To Do limits** -- In order to have more productive planning meetings, the number of tasks in the To Do section can be limited as well. The same as with WIP limits, it is written at the top of the To Do section or on top of the corresponding columns and limits the number of tasks in the To Do section or specific columns.

**The team**

Scrumban does not require any specific number of team members or team roles. The roles a team has prior to adopting Scrumban are kept when implementing Scrumban. They are reinforced by team members having to choose the tasks to complete themselves. The team roles in Scrumban are more specialized and less cross-functional than what is expected in scrum teams.

**Pull principle**

In Scrumban tasks are not assigned to the team members by the team leader or project manager. Each team member chooses which task from the To Do section they are going to complete next. This guarantees a smooth process flow, where all the team members are equally busy at all times.

**Feature freeze**

Feature freeze is used in Scrumban when the project deadline is approaching. It means that only the features that the team already has for development can still be worked on and no additional features can be added.[8]

**Triage**

Triage usually happens right after feature freeze. With an approaching project deadline, the project manager decides which of the in-development features will be completed and which will stay unfinished. This guarantees that the team can focus on finishing important features before the project deadline and forget the less important ones.[9]

**Terms**
Bucket size planning long-term planning approach in Scrumban, which is based on moving the plans through a few steps.

Lead and cycle time the time that is taken from task creation or beginning work on a task to its completion.

On demand planning planning technique that is executed only when there is a need for new tasks on the board.

Tooling

Like other methods, Scrumban can be implemented with the help of various tools. The most basic Scrumban implementation is a physical whiteboard with sticky notes. Electronic solutions, similar to scrum and kanban electronic boards are available as well. They offer a full automation of the board, where it only has to be updated by the team members. Electronic boards often also provide automatic reports, the possibility of attachments and discussions on tasks, time tracking, as well as integrations with other commonly used project management software.[10]

See also

- Kanban (development)
- List of software development philosophies
- Scrum (software development)

References


Shortest job next

Shortest job next (SJN), also known as shortest job first (SJF) or shortest process next (SPN), is a scheduling policy that selects for execution the waiting process with the smallest execution time.[1] SJN is a non-preemptive algorithm. Shortest remaining time is a preemptive variant of SJN.

Shortest job next is advantageous because of its simplicity and because it minimizes the average amount of time each process has to wait until its execution is complete. However, it has the potential for process starvation for processes which will require a long time to complete if short processes are continually added. Highest response ratio next is similar but provides a solution to this problem using a technique called aging.[2]

Another disadvantage of using shortest job next is that the total execution time of a job must be known before execution. While it is impossible to predict execution time perfectly, several methods can be used to estimate it, such as a weighted average of previous execution times.[3]

Shortest job next can be effectively used with interactive processes which generally follow a pattern of alternating between waiting for a command and executing it. If the execution burst of a process is regarded as a separate "job", past behaviour can indicate which process to run next, based on an estimate of its running time.

Shortest job next is used in specialized environments where accurate estimates of running time are available.

Contents

Weighted shortest job first
See also
References
External links

Weighted shortest job first

Weighted shortest job first (WSJF) is a modification of the concept used in agile development where jobs get weighted with the cost of delay so that the highest valued jobs get done sooner.[4]

Value-flow rate (VFR) is an alternate, more intuitive name given to WSJF which expresses cost of delay and duration using unitless relative "points" rather than actual units of time or money.[5]
See also

- Shortest remaining time

References


External links

- Scheduling Algorithm-SJF with example ([https://www.encious.com/SchedulingAlgorithmsSJF](https://www.encious.com/SchedulingAlgorithmsSJF))


This page was last edited on 28 January 2019, at 00:40 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.
Site Reliability Engineering

Site Reliability Engineering (SRE) is a discipline that incorporates aspects of software engineering and applies them to infrastructure and operations problems. The main goals are to create ultra-scalable and highly reliable software systems. According to Ben Treynor, founder of Google's Site Reliability Team, SRE is "what happens when a software engineer is tasked with what used to be called operations."[1]

Contents

History
Roles
DevOps vs SRE
See also
References
External links

History

Site Reliability Engineering was created at Google around 2003 when Ben Treynor was hired to lead a team of seven software engineers to run a production environment. The team was tasked to make Google's sites run smoothly, efficiently, and more reliably. Early on, Google's large-scale systems required the company to come up with new paradigms on how to manage such large systems and at the same time introduce new features continuously but at a very high-quality end user experience. The SRE footprint at Google is now larger than 1500 engineers. Many products have small to medium sized SRE teams supporting them, though not all products have SREs. The SRE processes that have been honed over the years are being used by other, mainly large scale, companies that are also starting to implement this paradigm. Intuit, ServiceNow, Microsoft, Apple, Twitter, Facebook, Dropbox, Amazon, Target, Dell Technologies, IBM, Xero, Oracle, Zalando, Acquia, VMware, GitHub, Waze, Home Depot, Capital One, and Ticketmaster have all put together SRE teams.

Roles

A site reliability engineer (SRE) will spend up to 50% of their time doing "ops" related work such as issues, on-call, and manual intervention. Since the software system that an SRE oversees is expected to be highly automatic and self-healing, the SRE should spend the other 50% of their time on development tasks such as new features, scaling or automation. The ideal SRE candidate is a highly skilled system administrator with knowledge of code and automation.

DevOps vs SRE

Coined around 2008, DevOps is a philosophy of cross team empathy and business alignment. It's also been associated with a practice that encompasses automation of manual tasks, continuous integration and continuous delivery. SRE and DevOps share the same foundational principles. SRE is viewed by many (as cited in the Google SRE book) as a "specific implementation of DevOps with some idiosyncratic extensions." SREs, being developers themselves, will naturally bring solutions that help remove the barriers between development teams and operations teams.
DevOps defines 5 key pillars of success:

1. Reduce organizational silos
2. Accept failure as normal
3. Implement gradual changes
4. Leverage tooling and automation
5. Measure everything

SRE satisfies the DevOps pillars as follows:[2]

1. Reduce organizational silos
   - SRE shares ownership with developers to create shared responsibility[3]
   - SREs use the same tools that developers use, and vice versa
2. Accept failure as normal
   - SREs embrace risk[4]
   - SRE quantifies failure and availability in a prescriptive manner using SLIs and SLOs[5]
   - SRE mandates blameless post mortems[6]
3. Implement gradual changes
   - SRE encourages developers and product owners to move quickly by reducing the cost of failure[4]
4. Leverage tooling and automation
   - SREs have a charter to automate menial tasks (called "toil") away[7]
5. Measure everything
   - SRE defines prescriptive ways to measure values[8]
   - SRE fundamentally believes that systems operation is a software problem

See also

- Cloud computing
- Data center
- High availability software
- Operations management
- Operations, administration and management
- Reliability engineering
- System administration

References

1. Are SRE the next data scientists? (https://techcrunch.com/2016/03/02/are-site-reliability-engineers-the-next-data-scientists), TechCrunch, Mar 2, 2016, Donald Fischer
landing.google.com.
landing.google.com.
landing.google.com.

General

- The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2, Thomas Limoncelli, ISBN 032194318X

External links

- Google SRE Books (https://landing.google.com/sre/books/)


This page was last edited on 5 March 2019, at 20:26 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.
Software development effort estimation

In software development, **effort estimation** is the process of predicting the most realistic amount of effort (expressed in terms of person-hours or money) required to develop or maintain software based on incomplete, uncertain and noisy input. Effort estimates may be used as input to project plans, iteration plans, budgets, investment analyses, pricing processes and bidding rounds.\[1\]

## State-of-practice

Published surveys on estimation practice suggest that expert estimation is the dominant strategy when estimating software development effort.\[2\]

Typically, effort estimates are over-optimistic and there is a strong over-confidence in their accuracy. The mean effort overrun seems to be about 30% and not decreasing over time. For a review of effort estimation error surveys, see.\[3\] However, the measurement of estimation error is problematic, see Assessing the accuracy of estimates. The strong overconfidence in the accuracy of the effort estimates is illustrated by the finding that, on average, if a software professional is 90% confident or “almost sure” to include the actual effort in a minimum-maximum interval, the observed frequency of including the actual effort is only 60-70%.\[4\]

Currently the term “effort estimate” is used to denote as different concepts such as most likely use of effort (modal value), the effort that corresponds to a probability of 50% of not exceeding (median), the planned effort, the budgeted effort or the effort used to propose a bid or price to the client. This is believed to be unfortunate, because communication problems may occur and because the concepts serve different goals.\[5\][6]

## History

Software researchers and practitioners have been addressing the problems of effort estimation for software development projects since at least the 1960s; see, e.g., work by Farr \[7][8\] and Nelson.\[9\]

Most of the research has focused on the construction of formal software effort estimation models. The early models were typically based on regression analysis or mathematically derived from theories from other domains. Since then a high number of model building approaches have been evaluated, such as approaches founded on case-based reasoning, classification and regression trees, simulation, neural networks, Bayesian statistics, lexical analysis of requirement specifications, genetic programming, linear programming, economic production models, soft computing, fuzzy logic modeling, statistical bootstrapping, and combinations of two or more of these models. The perhaps most common estimation methods today are the parametric estimation models COCOMO, SEER-SEM and SLIM. They have their basis in estimation research conducted in the 1970s and 1980s and are since then updated with new calibration data, with the last major release being COCOMO II in the year 2000. The estimation approaches based on functionality-based size measures, e.g., function points, is also based on research conducted in the 1970s and 1980s, but are re-calibrated with modified size measures and different counting approaches, such as the use case points \[10\] or object points in the 1990s and COSMIC (http://www.cosmic-sizing.org) in the 2000s.

## Estimation approaches

There are many ways of categorizing estimation approaches, see for example.\[11][12\] The top level categories are the following:

- **Expert estimation**: The quantification step, i.e., the step where the estimate is produced based on judgmental processes.\[13\]
- **Formal estimation model**: The quantification step is based on mechanical processes, e.g., the use of a formula derived from historical data.
- **Combination-based estimation**: The quantification step is based on a judgmental and mechanical combination of estimates from different sources.
Below are examples of estimation approaches within each category.

<table>
<thead>
<tr>
<th>Estimation approach</th>
<th>Category</th>
<th>Examples of support of implementation of estimation approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analogy-based estimation</td>
<td>Formal estimation model</td>
<td>ANGEL, Weighted Micro Function Points</td>
</tr>
<tr>
<td>WBS-based (bottom up) estimation</td>
<td>Expert estimation</td>
<td>Project management software, company specific activity templates</td>
</tr>
<tr>
<td>Parametric models</td>
<td>Formal estimation model</td>
<td>COCOMO, SLIM, SEER-SEM, TruePlanning for Software</td>
</tr>
<tr>
<td>Size-based estimation models[^14]</td>
<td>Formal estimation model</td>
<td>Function Point Analysis,[^15] Use Case Analysis, Use Case Points, SSU (Software Size Unit), Story points-based estimation in Agile software development, Object Points</td>
</tr>
<tr>
<td>Group estimation</td>
<td>Expert estimation</td>
<td>Planning poker, Wideband delphi</td>
</tr>
<tr>
<td>Mechanical combination</td>
<td>Combination-based estimation</td>
<td>Average of an analogy-based and a Work breakdown structure-based effort estimate[^16]</td>
</tr>
<tr>
<td>Judgmental combination</td>
<td>Combination-based estimation</td>
<td>Expert judgment based on estimates from a parametric model and group estimation</td>
</tr>
</tbody>
</table>

**Selection of estimation approaches**

The evidence on differences in estimation accuracy of different estimation approaches and models suggest that there is no “best approach” and that the relative accuracy of one approach or model in comparison to another depends strongly on the context.[^17] This implies that different organizations benefit from different estimation approaches. Findings[^18] that may support the selection of estimation approach based on the expected accuracy of an approach include:

- Expert estimation is on average at least as accurate as model-based effort estimation. In particular, situations with unstable relationships and information of high importance not included in the model may suggest use of expert estimation. This assumes, of course, that experts with relevant experience are available.
- Formal estimation models not tailored to a particular organization’s own context, may be very inaccurate. Use of own historical data is consequently crucial if one cannot be sure that the estimation model’s core relationships (e.g., formula parameters) are based on similar project contexts.
- Formal estimation models may be particularly useful in situations where the model is tailored to the organization’s context (either through use of own historical data or that the model is derived from similar projects and contexts), and it is likely that the experts’ estimates will be subject to a strong degree of wishful thinking.

The most robust finding, in many forecasting domains, is that combination of estimates from independent sources, preferable applying different approaches, will on average improve the estimation accuracy.^[18][19][20]

It is important to be aware of the limitations of each traditional approach to measuring software development productivity.^[21]

In addition, other factors such as ease of understanding and communicating the results of an approach, ease of use of an approach, and cost of introduction of an approach should be considered in a selection process.

**Assessing the accuracy of estimates**

The most common measure of the average estimation accuracy is the MMRE (Mean Magnitude of Relative Error), where the MRE of each estimate is defined as:

\[
MRE = \frac{|actual\ effort - estimated\ effort|}{actual\ effort}
\]

This measure has been criticized[^22][23][24] and there are several alternative measures, such as more symmetric measures[^25], Weighted Mean of Quartiles of relative errors (WMQ)[^26] and Mean Variation from Estimate (MVFE).[^27]

MRE is not reliable if the individual items are skewed. PRED(25) is preferred as a measure of estimation accuracy. PRED(25) measures the percentage of predicted values that are within 25 percent of the actual value.

A high estimation error cannot automatically be interpreted as an indicator of low estimation ability. Alternative, competing or complementing, reasons include low cost control of project, high complexity of development work, and more delivered functionality than originally estimated. A framework for improved use and interpretation of estimation error measurement is included in^[28].

**Psychological issues**

There are many psychological factors potentially explaining the strong tendency towards over-optimistic effort estimates that need to be dealt with to increase accuracy of effort estimates. These factors are essential even when using formal estimation models, because much of the input to these models is judgment-based. Factors that have been demonstrated to be important are: Wishful thinking, anchoring, planning fallacy and cognitive dissonance. A
discussion on these and other factors can be found in work by Jørgensen and Grimstad.\cite{29}

- It's easy to estimate what you know.
- It's hard to estimate what you know you don't know. (known unknowns)
- It's very hard to estimate things that you don't know you don't know. (unknown unknowns)

## Humor

The chronic underestimation of development effort has led to the coinage and popularity of numerous humorous adages, such as ironically referring to a task as a "small matter of programming" (when much effort is likely required), and citing laws about underestimation:

- **Ninety-ninety rule:**

  The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.\cite{30}

  — Tom Cargill, Bell Labs

- **Hofstadter's law:**

  Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

  — Douglas Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*\cite{31}

- **Fred Brooks' law:**

  What one programmer can do in one month, two programmers can do in two months.

  — Fred Brooks

Adding to the fact that estimating development efforts is hard, it's worth stating that assigning more resources doesn't always help.

## Comparison of development estimation software
<table>
<thead>
<tr>
<th>Software</th>
<th>Schedule estimate</th>
<th>Cost estimate</th>
<th>Cost Models</th>
<th>Input</th>
<th>Report Output Format</th>
<th>Supported Programming Languages</th>
<th>Platforms</th>
<th>Cost</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFCAA REVIC</td>
<td>Yes</td>
<td>Yes</td>
<td>REVIC</td>
<td>KLOC, Scale Factors, Cost Drivers</td>
<td>proprietary, Text</td>
<td>any</td>
<td>DOS</td>
<td>Free</td>
<td>Proprietary Free for public distribution</td>
</tr>
<tr>
<td>Seer for Software</td>
<td>Yes</td>
<td>Yes</td>
<td>SEER-SEM</td>
<td>SLOC, Function points, use cases, bottoms-up, object features</td>
<td>proprietary, Excel, Microsoft Project, IBM Rational, Oracle Crystal Ball</td>
<td>any</td>
<td>Windows, Any (Web-based)</td>
<td>Commercial</td>
<td>Proprietary</td>
</tr>
<tr>
<td>SLIM</td>
<td>Yes</td>
<td>Yes</td>
<td>SLIM</td>
<td>Size (SLOC, Function points, Use Cases, etc.), constraints (size, duration, effort, staff), scale factors, historical projects, historical trends</td>
<td>proprietary, Excel, Microsoft Project, Microsoft PowerPoint, IBM Rational, text, HTML</td>
<td>any</td>
<td>Windows, Any (Web-based)</td>
<td>Commercial</td>
<td>Proprietary</td>
</tr>
<tr>
<td>TruePlanning</td>
<td>Yes</td>
<td>Yes</td>
<td>PRICE</td>
<td>Components, Structures, Activities, Cost drivers, Processes, Functional Software Size (Source Lines of Code (SLOC), Function Points, Use Case Conversion Points (UCCP), Predictive Object Points (POPs) etc.)</td>
<td>Excel, CAD</td>
<td>any</td>
<td>Windows</td>
<td>Commercial</td>
<td>Proprietary</td>
</tr>
</tbody>
</table>

See also

- Cone of uncertainty
- Cost estimation in software engineering
- Cost estimation models
- Cost overrun
- Function points
- Planning fallacy
- Proxy-based estimating
- Putnam model
- Software metric
- Software parametric models

References

Technology roadmap

A technology roadmap is a flexible planning technique to support strategic and long-range planning, by matching short-term and long-term goals with specific technology solutions.\[1\][2] It is a plan that applies to a new product or process and may include using technology forecasting/technology scouting to identify suitable emerging technologies.\[3\] It is a known technique to help manage the fuzzy front-end of innovation.\[4\] It is also expected that roadmapping techniques may help companies to survive in turbulent environments\[1\] and help them to plan in a more holistic way to include non-financial goals and drive towards a more sustainable development.\[5\] Here roadmaps can be combined with other corporate foresight methods to facilitate systemic change.\[6\]

Developing a roadmap has three major uses.\[7\] It helps reach a consensus about a set of needs and the technologies required to satisfy those needs, it provides a mechanism to help forecast technology developments, and it provides a framework to help plan and coordinate technology developments.\[8\] It may also be used as an analysis tool to map the development and emergence from new industries.

### Contents

- **Process**
  - Phase 1: Preliminary phase
  - Phase 2: Development phase
  - Phase 3: Follow-up activity phase

- **The fast-start approach to roadmapping**

- **Planning and business development context**
  - Knowledge and skills required
  - Purpose

- **Case studies**

- **See also**

- **References**

- **Further reading**

### Process

The technology roadmapping process may be conducted in three phases (see figure 1): preliminary activities, the development of the roadmap, and the follow-up activities phase. Because the process is too big for one model, the phases are modeled separately. In the models no different roles are made; this is because everything is done by the participants as a group.\[9\]
Phase 1: Preliminary phase

The first phase, the preliminary phase (see figure 2), consists of 3 steps:

1. satisfy essential conditions,
2. provide leadership / sponsorship, and
3. define the scope and boundaries for the technology roadmap.

In this phase the key decision makers must identify that they have a problem and that technology roadmapping can help them in solving the problem.

Satisfy essential conditions

In this step it must become clear what the conditions are (they must be identified) and if they are not met, who takes actions to meet them. These conditions include, for example:

- A need for the technology roadmap
- Input and participation from different parts of the organization (e.g., marketing, R&D, the strategic business units) with different planning horizons and perspectives.

All conditions should be satisfied (or an agreed-on party takes necessary actions) to continue to the next step. The participants can have zero or more conditions of their own. It applies to all conditions that have the attribute to be met or not.

Provide leadership / sponsorship

Committed leadership is needed because of the time and effort involved in creating a technology roadmap. Additionally the leadership should come from one of the participants, one of them provides leadership and sponsorship. This means that the line organization must drive the process and use the roadmap to make resource allocation decisions.\cite{10}

Define the scope and boundaries

In this step the context for the roadmap is specified. In the company a vision should exist and it must be clear that the roadmap can support that vision. If the vision does not exist one should be developed and clearly stated. When that is done the boundaries and the scope of the roadmap should be specified. Furthermore, the planning horizon and the level of details should be set. The scope can be further divided into the technology scope and the participation scope.

In table 1 all the different sub-activities of the preliminary activity phase can be seen. All the sub-activities have concepts as end products (marked in **bold**). These concepts are the actual meta-data model, which is an adjusted class diagram.\cite{11}
Table 1. Activity table for the preliminary activity phase

<table>
<thead>
<tr>
<th>Activity</th>
<th>Sub-Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Satisfy essential conditions</td>
<td>Identify essential conditions</td>
<td>When all the <strong>participants</strong> come together, essential <strong>conditions</strong> can be identified (e.g. what groups should be involved, what are the key customers and what are the key suppliers).</td>
</tr>
<tr>
<td></td>
<td>Take action to satisfy conditions</td>
<td>For technology roadmapping to succeed, <strong>conditions</strong> from the <strong>participants</strong> must be satisfied.</td>
</tr>
<tr>
<td>Provide leadership / sponsorship</td>
<td>Clearly state vision</td>
<td>The already existing <strong>vision</strong> must be clear.</td>
</tr>
<tr>
<td>Define the scope and boundaries for the technology roadmap</td>
<td>Develop vision</td>
<td>The <strong>vision</strong> is developed and stated clearly.</td>
</tr>
<tr>
<td></td>
<td>Define scope</td>
<td>The <strong>scope</strong> of the project can further define the set of <strong>needs</strong>, <strong>planning horizon</strong> and level of detail. The scope can be further divided into the <strong>technology scope</strong> and the <strong>participation scope</strong>.</td>
</tr>
<tr>
<td></td>
<td>Define boundaries</td>
<td>The <strong>boundaries</strong> should also be included.</td>
</tr>
</tbody>
</table>

**Phase 2: Development phase**

The second phase, the development of the technology roadmap phase (see figure 3.), consists of 7 steps:

1. Identify the "product" that is the focus of the roadmap,
2. Identify the critical system requirements and their targets,
3. Specify the major technology areas,
4. Specify the technology drivers and their targets,
5. Identify technology alternatives and their timelines,
6. Recommend the technology alternatives that should be pursued, and
7. create the technology roadmap report.

— Defense Logistics Agency, [12]

**Identify the product focus of the roadmap**

In this step the common product needs are identified and are agreed on by all the participants. This is important to get the acceptance of all groups for the process. In case of uncertainty of the product needs scenario-based planning can be used to determine the common product needs. In figure 3, the participants and possibly the scenario-based planning provide the
common product needs.

**Identify the critical system requirements and their targets**

Once it is decided what must be roadmapped, the critical system requirements can be identified; they provide the overall framework for the technology roadmap. The requirements can have targets (as an attribute in figure 3) like reliability and costs.

**Specify the major technology areas**

These are the areas that help achieve critical system requirements. For each technology area several technologies can be found. Example technology areas are: market assessment, crosscutting technology, component development, and system development.

**Specify the technology drivers and their targets**

In this step the critical system requirements from the second step are transformed into technology drivers (with targets) for the specific technology area. These drivers are the critical variables that select the technology alternatives. Drivers depend on the technology areas but they relate to how the technology addresses the critical system requirements.

**Identify technology alternatives and their timelines**

At this point the technology drivers and their targets are specified and the technology alternatives that can satisfy those targets should be specified. For each of the alternatives a timeline should be estimated for how it will mature with respect to the technology driver targets.

The time factor can be adapted suitable for the particular situation. The time horizons for e-commerce and software related sectors are usually short. Other distinctions can be made on scale and intervals.

**Recommend the technology alternatives that should be pursued**

Because the alternatives may differ in costs, timeline, etc., a selection must be made of the alternatives. These are the alternatives to pursue in figure 3. In this step a lot of trade-offs must be made between different alternatives for different targets: for example, performance over costs and even target over target.

**Create the report**

At this point the technology roadmap is finished. In figure 3, it can be seen that the technology roadmap report consists of 5 parts:

1. the identification and description of each technology area,
2. critical factors in the roadmap,
3. unaddressed areas,
4. implementation recommendations, and
5. technical recommendations.

The report can also include additional information. In table 2 all the different sub-activities of the development phase can be seen.
Phase 3: Follow-up activity phase

This is the moment when the roadmap must be critiqued, validated and hopefully accepted by the group involved in any implementation. This requires a plan developed using the technology roadmap. Next, there must be a periodical review and update point, because needs from the participants and the technologies evolve.

The fast-start approach to roadmapping

Given the potential complexity and organisational inertia surrounding the creation of roadmaps, researchers at the University of Cambridge[13] focused on developing a fast-start approach to roadmapping.[14] This approach, called T-Plan, was created in the late 1990s primarily to help organisations take the first step into roadmapping with minimal resource and time commitment. It has been influential in the propagation and uptake of roadmapping internationally including translations of the T-Plan workbook[15] into Chinese (traditional & modern), German, Japanese and Spanish. The approach (as well as its counterpart for innovation and strategy roadmapping, S-Plan) is flexible and scalable, and
therefore can be easily customised for efficient application.\textsuperscript{[16]} Fast and lean approaches are particularly important for small and medium enterprises (SME) and have been shown to work in particularly to give directions to clusters of SMEs.\textsuperscript{[17]}

**Planning and business development context**

The process of technology roadmapping fits into corporate strategy, corporate strategic planning, technology planning and the business development context. Three critical elements should be connected: needs, products, and technology.

**Knowledge and skills required**

**Consultant with skills**

Creating a technology roadmap requires certain knowledge and skills. Some of the participants must know the purpose of technology roadmapping. Next to this group-process and interpersonal skills are required since the process includes a lot of discussions and finding out what the common need is. If the number of participants is really large there might be need for a consultant or facilitator.

**Purpose**

**Product planning in roadmapping**

This is the most common type of a technology roadmap: linking the insertion of technologies into products.

**Programme planning**

This type is more directed to the implementation of strategy and related to project planning. Figure 5 shows the relationships between technology development phases, programme phases and milestones.

**Formats**

- Bars: Almost all the roadmaps are (partly) expressed in bars for each layer. This makes the roadmaps very simple and unified, which makes the communication and integration easier.
- Graphs: A technology roadmap can also be expressed as a graph, usually one for each of the sub layers. (e.g. IMEC uses the second method).

**Case studies**

Documented case studies include:

- ABB\textsuperscript{[18]}
- Mining\textsuperscript{[19]}
- Motorola\textsuperscript{[20]}

\textsuperscript{[16]}\textsuperscript{[17]}\textsuperscript{[18]}\textsuperscript{[19]}\textsuperscript{[20]}

Business plan  
Enterprise systems engineering  
Information technology planning  
Project network  
Requirement prioritization  
Strategic management  
Strategic technology plan  
Technology life cycle  
Work breakdown structure

References

3. Garcia97
Further reading


The Mythical Man-Month: Essays on Software Engineering is a book on software engineering and project management by Fred Brooks first published in 1975, with subsequent editions in 1982 and 1995. Its central theme is that "adding manpower to a late software project makes it later". This idea is known as Brooks' law, and is presented along with the second-system effect and advocacy of prototyping.

Brooks' observations are based on his experiences at IBM while managing the development of OS/360. He had added more programmers to a project falling behind schedule, a decision that he would later conclude had, counter-intuitively, delayed the project even further. He also made the mistake of asserting that one project—involving in writing an ALGOL compiler—would require six months, regardless of the number of workers involved (it required longer). The tendency for managers to repeat such errors in project development led Brooks to quip that his book is called "The Bible of Software Engineering", because "everybody quotes it, some people read it, and a few people go by it". The book is widely regarded as a classic on the human elements of software engineering.

### Contents

#### Editions

<table>
<thead>
<tr>
<th>Ideas presented</th>
</tr>
</thead>
<tbody>
<tr>
<td>The mythical man-month</td>
</tr>
<tr>
<td>No silver bullet</td>
</tr>
<tr>
<td>The second-system effect</td>
</tr>
<tr>
<td>The tendency towards irreducible number of errors</td>
</tr>
<tr>
<td>Progress tracking</td>
</tr>
<tr>
<td>Conceptual integrity</td>
</tr>
<tr>
<td>The manual</td>
</tr>
<tr>
<td>The pilot system</td>
</tr>
<tr>
<td>Formal documents</td>
</tr>
<tr>
<td>Project estimation</td>
</tr>
<tr>
<td>Communication</td>
</tr>
<tr>
<td>The surgical team</td>
</tr>
<tr>
<td>Code freeze and system versioning</td>
</tr>
<tr>
<td>Specialized tools</td>
</tr>
<tr>
<td>Lowering software development costs</td>
</tr>
</tbody>
</table>

#### See also

- Bibliography
- References
- External links

---

**Author** | Frederick Brooks  
**Subject** | Software project management  
**Publisher** | Addison-Wesley  
**Publication date** | 1975, 1995  
**OCLC** | 1201368 (https://www.worldcat.org/oclc/1201368)  
**Dewey Decimal** | 001.6/425  
**LC Class** | QA76.6 .B75  
**Followed by** | "No Silver Bullet"
Editions


Ideas presented

The mythical man-month

Brooks discusses several causes of scheduling failures. The most enduring is his discussion of Brooks's law: *Adding manpower to a late software project makes it later*. Man-month is a hypothetical unit of work representing the work done by one person in one month; Brooks' law says that the possibility of measuring useful work in man-months is a myth, and is hence the centerpiece of the book.

Complex programming projects cannot be perfectly partitioned into discrete tasks that can be worked on without communication between the workers and without establishing a set of complex interrelationships between tasks and the workers performing them.

Therefore, assigning more programmers to a project running behind schedule will make it even later. This is because the time required for the new programmers to learn about the project and the increased communication overhead will consume an ever increasing quantity of the calendar time available. When *n* people have to communicate among themselves, as *n* increases, their output decreases and when it becomes negative the project is delayed further with every person added.

- Group intercommunication formula: \( n(n - 1) / 2 \)
- Example: 50 developers give \( 50 \cdot (50 - 1) / 2 = 1225 \) channels of communication.

No silver bullet

Brooks added "*No Silver Bullet — Essence and Accidents of Software Engineering*"—and further reflections on it, "*No Silver Bullet' Refired*"—to the anniversary edition of *The Mythical Man-Month*.

Brooks insists that there is no one silver bullet -- "there is no single development, in either technology or management technique, which by itself promises even one order of magnitude [tenfold] improvement within a decade in productivity, in reliability, in simplicity."

The argument relies on the distinction between accidental complexity and essential complexity, similar to the way Amdahl's law relies on the distinction between "strictly serial" and "parallelizable".

The second-system effect

The second-system effect proposes that, when an architect designs a second system, it is the most dangerous system they will ever design, because they will tend to incorporate all of the additions they originally did not add to the first system due to inherent time constraints. Thus, when embarking on a second system, an engineer should be mindful that they are susceptible to over-engineering it.
The tendency towards irreducible number of errors

The author makes the observation that in a suitably complex system there is a certain irreducible number of errors. Any attempt to fix observed errors tends to result in the introduction of other errors.

Progress tracking

Brooks wrote "Question: How does a large software project get to be one year late? Answer: One day at a time!" Incremental slippages on many fronts eventually accumulate to produce a large overall delay. Continued attention to meeting small individual milestones is required at each level of management.

Conceptual integrity

To make a user-friendly system, the system must have conceptual integrity, which can only be achieved by separating architecture from implementation. A single chief architect (or a small number of architects), acting on the user's behalf, decides what goes in the system and what stays out. The architect or team of architects should develop an idea of what the system should do and make sure that this vision is understood by the rest of the team. A novel idea by someone may not be included if it does not fit seamlessly with the overall system design. In fact, to ensure a user-friendly system, a system may deliberately provide fewer features than it is capable of. The point being, if a system is too complicated to use, many features will go unused because no one has time to learn them.

The manual

The chief architect produces a manual of system specifications. It should describe the external specifications of the system in detail, i.e., everything that the user sees. The manual should be altered as feedback comes in from the implementation teams and the users.

The pilot system

When designing a new kind of system, a team will design a throw-away system (whether it intends to or not). This system acts as a "pilot plan" that reveals techniques that will subsequently cause a complete redesign of the system. This second, smarter system should be the one delivered to the customer, since delivery of the pilot system would cause nothing but agony to the customer, and possibly ruin the system's reputation and maybe even the company.

Formal documents

Every project manager should create a small core set of formal documents defining the project objectives, how they are to be achieved, who is going to achieve them, when they are going to be achieved, and how much they are going to cost. These documents may also reveal inconsistencies that are otherwise hard to see.

Project estimation

When estimating project times, it should be remembered that programming products (which can be sold to paying customers) and programming systems are both three times as hard to write as simple independent in-house programs.[4] It should be kept in mind how much of the work week will actually be spent on technical issues, as opposed to administrative or other non-technical tasks, such as meetings, and especially "stand-up" or "all-hands" meetings.
**Communication**

To avoid disaster, all the teams working on a project should remain in contact with each other in as many ways as possible — e-mail, phone, meetings, memos etc. Instead of assuming something, implementers should ask the architect(s) to clarify their intent on a feature they are implementing, before proceeding with an assumption that might very well be completely incorrect. The architect(s) are responsible for formulating a group picture of the project and communicating it to others.

**The surgical team**

Much as a surgical team during surgery is led by one surgeon performing the most critical work, while directing the team to assist with less critical parts, it seems reasonable to have a "good" programmer develop critical system components while the rest of a team provides what is needed at the right time. Additionally, Brooks muses that "good" programmers are generally five to ten times as productive as mediocre ones.

**Code freeze and system versioning**

Software is invisible. Therefore, many things only become apparent once a certain amount of work has been done on a new system, allowing a user to experience it. This experience will yield insights, which will change a user's needs or the perception of the user's needs. The system should, therefore, be changed to fulfill the changed requirements of the user. This can only occur up to a certain point, otherwise the system may never be completed. At a certain date, no more changes should be allowed to the system and the code should be frozen. All requests for changes should be delayed until the next version of the system.

**Specialized tools**

Instead of every programmer having his own special set of tools, each team should have a designated tool-maker who may create tools that are highly customized for the job that team is doing, e.g., a code generator tool that creates code based on a specification. In addition, system-wide tools should be built by a common tools team, overseen by the project manager.

**Lowering software development costs**

There are two techniques for lowering software development costs that Brooks writes about:

- Implementers may be hired only after the architecture of the system has been completed (a step that may take several months, during which time prematurely hired implementers may have nothing to do).
- Another technique Brooks mentions is not to develop software at all, but simply to buy it "off the shelf" when possible.

**See also**

- Anti-pattern
- Code refactoring
- *Peopleware: Productive Projects and Teams*
- Software development process
- Hofstadter's law

**Bibliography**


References

3. This humorous song based on 99 Bottles of Beer has been around on notice boards since at least 2000 (Anonymous (2000). "Computer programming quotes" (http://www.gdargaud.net/Humor/QuotesProgramming.html).
4. Mythical Man Month (https://www.amazon.com/gp/reader/0201835959) Figure 1.1, Page 13

External links

- Frederick P. Brooks, Jr. Homepage (http://www.cs.unc.edu/~brooks/)
- Preface to the First Edition, as found on Safari.Informit.com (http://safari.informit.com/0201835959/pref03#X2ludGVyb mFsX1RvYz94bWxpZD0wMjAxOTU5L3ByZWYwMg==)
- Organization and Team Patterns (http://www.dfpug.de/loseblattsammlung/online/workshop/design_patterns/sonstiges.htm)
- A review by Hector Correa on chapters "The Mythical Man-Month" and "No Silver Bullet – Essence and Accident" (http://www.hectorcorrea.com/blog/The-Mythical-Man-Month.aspx)
- Selected TEXT from The Mythical Man-Month (https://sites.google.com/site/themythicalmanmonth/home/)
- Full text of 1975 edition (archive.org) (https://archive.org/details/mythicalmanmonth00fred)


This page was last edited on 31 August 2018, at 14:00 (UTC).
In time management, **timeboxing** allocates a fixed time period, called a **timebox**, within which planned activity takes place. It is employed by several project management approaches and for personal time management.

**Contents**

**In project management**
- As an alternative to fixing scope
- To manage risk
- Adoption in software development

**In personal time management**
- Relationship with other methods
- References
- External links

**In project management**

Timeboxing is used as a project planning technique. The schedule is divided into a number of separate time periods (timeboxes), with each part having its own deliverables, deadline and budget. Sometimes referred to as *schedule as independent variable* (SAIV).[^2]

**As an alternative to fixing scope**

In project management, there are generally considered to be three constraints of time (sometimes schedule), cost (sometimes budget), and scope;[^3][^4][^5][^6][^7] with quality often added as a fourth constraint (represented as the middle of a triangle).[^8][^9][^10] The assumption is that a change in one constraint will affect the others.[^6]

Without timeboxing, projects usually work to a fixed scope,[^11] in which case when it becomes clear that some deliverables cannot be completed within the planned timescales, either the deadline has to be extended (to allow more time to complete the fixed scope) or more people are involved (to complete the fixed scope in the same time). Often both happen, resulting in delayed delivery, increased costs, and often reduced quality (as per The Mythical Man-Month principle).

With timeboxing, the deadline is fixed, meaning that the scope would have to be reduced. As this means organizations have to focus on completing the most important deliverables first, timeboxing often goes hand-in-hand with a scheme for prioritizing of deliverables (such as with the MoSCoW method).[^12]

**To manage risk**

Timeboxes are used as a form of risk management, to explicitly identify uncertain task/time relationships, i.e., work that may easily extend past its deadline. Time constraints are often a primary driver in planning and should not be changed without considering project or sub-project critical paths. That is, it’s usually important to meet deadlines. Risk factors for missed deadlines can include complications upstream of the project, planning errors within the project, team-related...
issues, or faulty execution of the plan. Upstream issues might include changes in project mission or backing/support from management. A common planning error is inadequate task breakdown, which can lead to underestimation of the time required to perform the work. Team-related issues can include trouble with inter-team communication; lack of experience or required cross-functionality; lack of commitment/drive/motivation (i.e. poor team building and management).

To stay on deadline, the following actions against the triple constraints are commonly evaluated:

- Reduce scope: drop requirements of lower impact (the ones that will not be directly missed by the user)
- Time is the fixed constraint here
- Increase cost: e.g., add overtime or resources

**Adoption in software development**

Many successful software development projects use timeboxing, especially smaller ones. Adopting timeboxing more than tripled developer productivity at DuPont in the '80s. In some cases, applications were completely delivered within the time estimated to complete just a specification. However, Steve McConnell argues that not every product is suitable and that timeboxing should only be used after the customer agrees to cut features, not quality. There is little evidence for strong adoption amongst the largest class of projects.

Timeboxing has been adopted by some notable software development methodologies:

- Dynamic systems development method (DSDM)
- In lean software development, pull scheduling with Kanban provides short term time management. When developing a large and complex system, when long term planning is required timeboxing is layered above.
- Rapid application development (RAD) software development process features iterative development and software prototyping. According to Steve McConnell, timeboxing is a "Best Practice" for RAD and a typical timebox length should be 60–120 days.
- Scrum was influenced by ideas of timeboxing and iterative development. Regular timeboxed units known as sprints form the basic unit of development. A typical length for a sprint is less than 30 days. Sprint planning, sprint retrospective and sprint review meetings are timeboxed.
- In Extreme programming methodologies, development planning is timeboxed into iterations typically 1, 2 or 3 weeks in length. The business revalues pending user stories before each iteration.

Agile software development advocates moving from plan-driven to value-driven development. Quality and time are fixed but flexibility allowed in scope. Delivering the most important features first leads to an earlier return on investment than the waterfall model.

A lack of detailed specifications typically is the result of a lack of time, or the lack of knowledge of the desired end result (solution). In many types of projects, and especially in software engineering, analyzing and defining all requirements and specifications before the start of the realization phase is impossible. Timeboxing can be a favorable type of contracting for projects in which the deadline is the most critical aspect and when not all requirements are completely specified up front. This also allows for new feedback or insights discovered during the project to be reflected in the end result.

**In personal time management**

Timeboxing can be used for personal tasks, as well, in which case it uses a reduced scale of time (e.g., thirty minutes) and of deliverables (e.g., a household chore instead of project deliverable).

Personal timeboxing is also said to act as a life hack to help curb perfectionist tendencies (by setting a firm time and not overcommitting to a task) which can also enhance creativity and focus (by creating a sense of urgency or increased pressure).
Relationship with other methods

Timeboxing acts as a building block in other personal time management methods:

- The Pomodoro Technique is based on 25 minute timeboxes of focused concentration separated by breaks allowing the mind to recover.[23]
- Andy Hunt gives timeboxing as his 'T' in SMART.[24]

References

11. Godin, Seth. Getting Real: The smarter, faster, easier way to build a successful web application. 37signals.
How To Use Timeboxing To Get More Done (http://www.hypnobusters.com/blog/how-to-use-timeboxing-to-get-more-done/)
Agile Requirements Change Management (http://www.agilemodeling.com/essays/changeManagement.htm)
Designing To Schedule (http://hosteddocs.ittoolbox.com/EM032006.pdf)
Using timeboxing for a motivation boost (http://motivationgrid.com/timebox-quick-motivation-boost)


This page was last edited on 15 February 2019, at 19:27 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.
User experience design

User experience design (UXD, UED, or XD) is the process of enhancing user satisfaction with a product by improving the usability, accessibility, and desirability provided in the interaction with a product.[1] User experience design encompasses traditional human–computer interaction (HCI) design, and extends it by addressing all aspects of a product or service as perceived by users.[2]

Contents

History

Elements
- Visual design
- Information architecture
  - Structuring, organization, and labeling
  - Finding and managing
- Interaction design
- Usability
- Usability testing
- Accessibility
  - WCAG compliance
- Human–computer interaction

Design
- General design process

UX Deliverables

Designers
- Graphic designers
- Visual designers
- Interaction designers

Testing the design

Benefits

See also

References

Further reading

History

The field of user experience design is a conceptual design discipline and has its roots in human factors and ergonomics, a field that, since the late 1940s, has focused on the interaction between human users, machines, and the contextual environments to design systems that address the user's experience.[3] With the proliferation of workplace computers in the early 1990s, user experience started to become a concern for designers. Donald Norman, a professor and researcher in design, usability, and cognitive science, coined the term "user experience", and brought it to a wider audience.[4]
I invented the term because I thought human interface and usability were too narrow. I wanted to cover all aspects of the person's experience with the system including industrial design graphics, the interface, the physical interaction and the manual. Since then the term has spread widely, so much so that it is starting to lose its meaning.

— Donald Norman[5]

Elements

User experience design draws from design approaches like human-computer interaction and user-centered design and includes elements from similar disciplines like interaction design, visual design, information architecture, user research, and others.

Visual design

Visual design, also commonly known as graphic design, user interface design, communication design, and visual communication, represents the aesthetics or look-and-feel of the front end of any user interface. Graphic treatment of interface elements is often perceived as the visual design. The purpose of visual design is to use visual elements like colors, images, and symbols to convey a message to its audience. Fundamentals of Gestalt psychology and visual perception give a cognitive perspective on how to create effective visual communication.[6]

Information architecture

Information architecture is the art and science of structuring and organizing the information in products and services to support usability and findability.

In the context of information architecture, information is separate from both knowledge and data, and lies nebulously between them. It is information about objects.[7] The objects can range from websites, to software applications, to images et al. It is also concerned with metadata: terms used to describe and represent content objects such as documents, people, process, and organizations. Information Architect also encompasses how the pages and navigation are structured.

Structuring, organization, and labeling

Structuring is reducing information to its basic building units and then relating them to each other. Organization involves grouping these units in a distinctive and meaningful hierarchy. Labeling means using appropriate wording and nomenclature to support easy navigation and findability.

Finding and managing

Findability is the most critical success factor for information architecture. If users are not able to find required information without browsing, searching or asking, then the find-ability of the information architecture fails. Navigation needs to be clearly conveyed to ease finding of the contents.

Interaction design

It is well recognized that component of interaction design is an essential part of user experience (UX) design, centering on the interaction between users and products [8]. The goal of interaction design is to create a product that produces an efficient and delightful end-user experience by enabling users to achieve their objectives in the best way possible[9] [10]
The current high emphasis on user-centered design and the strong focus on enhancing user experience have made interaction designers critical in conceptualizing products to match user expectations and meet the standards of the latest UI patterns and components.[11] To enable a pleasurable and desirable end user experience, the following are some considerations for the interaction design process:

- Defining interaction patterns best suited in the context
- Incorporating user needs collected during user research into the designs
- Features and information that are important to the user
- Interface behavior like drag-drop, selections, and mouse-over actions
- Effectively communicating strengths of the system
- Making the interface intuitive by building affordances
- Maintaining consistency throughout the system.
- Utilizing a haptic feedback system to reduce confusion[12]

In the last few years, the role of interaction designer has shifted from being just focused on specifying UI components and communicating them to the engineers to a situation now where designers have more freedom to design contextual interfaces which are based on helping meet the user needs.[13] Therefore, User Experience Design evolved into a multidisciplinary design branch that involves multiple technical aspects from motion graphics design and animation to programming.

**Usability**

Usability is the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.[14]

Usability is attached with all tools used by humans and is extended to both digital and non-digital devices. Thus, it is a subset of user experience but not wholly contained. The section of usability that intersects with user experience design is related to humans' ability to use a system or application. Good usability is essential to a positive user experience but does not alone guarantee it.[15]

**Usability testing**

Usability testing is a technique used in user-centered interaction design to evaluate a product by testing it on users. This can be seen as an irreplaceable usability practice, since it gives direct input on how real users use the system. It is a measure of how fast a user can perform given tasks to test the efficiency and intuitiveness of a product.

**Accessibility**

Accessibility of a system describes its ease of reach, use and understanding. In terms of user experience design, it can also be related to the overall comprehensibility of the information and features. It helps shorten the learning curve associated with the system. Accessibility in many contexts can be related to the ease of use for people with disabilities and comes under usability.[16]

**WCAG compliance**

Web Content Accessibility Guidelines (WCAG) 2.0 covers a wide range of recommendations for making Web content more accessible. Following these guidelines will make content accessible to a wider range of people with disabilities, including blindness and low vision, deafness and hearing loss, learning disabilities, cognitive limitations, limited movement, speech
disabilities, photosensitivity and combinations of these. Following these guidelines will also often make Web content more usable to users in general.\[17\] Making content more usable and readily accessible to all types of users enhances a user's overall user experience.

**Human–computer interaction**

Human–computer interaction is concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them.\[18\]

Human–computer interaction is the main contributor to user experience design because of its emphasis on human performance rather than mere usability. It provides key research findings which inform the improvement of systems for the people. Human-computer interaction extends its study towards more integrated interactions, such as tangible interactions, which is generally not covered in the practice of user experience. User experience cannot be manufactured or designed; it has to be incorporated in the design. Understanding the user's emotional quotient plays a key role while designing a user experience. The first step while designing the user experience is determining the reason a visitor will be visiting the website or use the application in question. Then the user experience can be designed accordingly.

**Design**

User experience design incorporates most or all of the above disciplines to positively impact the overall experience a person has with a particular interactive system and its provider. User experience design most frequently defines a sequence of interactions between a user (individual person) and a system, virtual or physical, designed to meet or support user needs and goals, primarily, while also satisfying systems requirements and organizational objectives.

Typical outputs include:

- Persona (an archetypal user for whom the product or service is being designed)
- Wireframes (screen blueprints or storyboards)
- Prototypes (for interactive or in-the-mind simulation)
- Written specifications (describing the behavior or design), e.g. use cases
- Site audit (usability study of existing assets)
- User-Flow diagrams and navigation maps
- User stories or scenarios
- Sitemaps and content inventory
- High-fidelity visual mockups (precise visual layout and design of the expected product or interface)

**General design process**

While designing a product or service for a client, it is of utmost importance that the designers are on the same page as the client. All the information collected, plans made, design executed will reflect on the final product. Rigorous analysis must be done before proceeding to the design stage and then numerous testings done to optimize the site as per best standards so that the competitive edge is maintained. Leading Digital marketing companies combine three elements to provide the best responsive product to the customer. These are:

1. Researching about the target audience
2. Understanding the company's business goals
3. And most importantly apply out of the box thinking.

Brainstorming and testing ultimately leads them to finalize the design for their customers. Let's have a detailed look at the step by step process of product design:
- **Collecting information about the problem**

The UX designer needs to find out as much as they can about people, processes, and products before the design phase. Designers can do this by meeting with the clients or business stakeholders frequently to know what their requirements are, or by conducting interviews with users in their home or work spaces. This kind of qualitative research helps designers create products and services that better serve user needs.

- **Getting ready to design**

After research, the designer must make sense of the data they've collected. Typically this is done through modeling of the users and their environments. User modeling or personas are composite archetypes based on behavior patterns uncovered during research. Personas provide designers a precise way of thinking and communicating about how groups of users behave, how they think, what they want to accomplish and why.[19] Once created, personas help the designer to understand the users' goals in specific contexts, which is particularly useful during ideation and for validating design concepts. Other types of models include work flow models, artifact models, and physical models.

- **Design**

When the designer has a firm grasp on the user's needs and goals, they begin to sketch out the interaction framework (also known as wireframes). This stage defines the high-level structure of screen layouts, as well as the product's flow, behavior, and organization. There are many kinds of materials that can be involved in during this iterative phase, from whiteboards to paper prototypes. As the interaction framework establishes an overall structure for product behavior, a parallel process focused on the visual and industrial designs. The visual design framework defines the experience attributes, visual language, and the visual style.[20]

Once a solid and stable framework is established, wireframes are translated from sketched storyboards to full-resolution screens that depict the user interface at the pixel level. At this point, it's critical for the programming team to collaborate closely with the designer. Their input is necessary to creating a finished design that can and will be built while remaining true to the concept.

- **Test and iterate**

Usability testing is carried out through prototypes (paper or digital). The target users are given various tasks to perform on the prototypes. Any issues or problems faced by the users are collected as field notes and these notes are used to make changes in the design and reiterate the testing phase.[21] Usability testing is, at its core, a means to "evaluate, not create".[22]

---

**UX Deliverables**

UX designers’ main goal is to solve the end-users' problems, and thus the ability to communicate the design to stakeholders and developers is critical to the ultimate success of the design. Regarding UX specification documents, these requirements depend on the client or the organization involved in designing a product. The four major deliverables are: a title page, an introduction to the feature, wireframes and a version history.[23] Depending on the type of project, the specification documents can also include flow models, cultural models, personas, user stories, scenarios and any prior user research. Documenting design decisions, in the form of annotated wireframes, gives the developer the necessary information they may need to successfully code the project.

Depending on the company, a user experience designer may need to be a jack of all trades. It is not uncommon to see a user experience designer jump in at the beginning of the project lifecycle, where the problem set and project definition is vague, or after the project requirements document has been finalized and wireframes and functional annotations need to be created.
The following details the responsibilities a user experience designer may have at each phase of a project:

**At the beginning, when the project is more conceptual:**
- Ethnographic research
- Surveying
- Customer feedback and testing
- Focus group administration
- Non-directed interview
- Contextual Interview
- Mental modeling
- Flow charts
- Mood boards
- Card sorting
- Competitive analysis
- Contextual Inquiry

**While the project is underway:**
- Wireframing
- Heuristic analysis
- Expert evaluation
- Pluralistic walkthrough
- Personas
- Scenario
- Prototypes
- System mapping
- Experience mapping
- User testing/usability testing

**After the project has launched:**
- User testing/usability testing
- A/B testing
- Additional wireframing as a result of test results and fine-tuning

---

**Designers**

As with the fields mentioned above, user experience design is a highly multi-disciplinary field, incorporating aspects of psychology, anthropology, architecture, sociology, computer science, graphic design, industrial design, cognitive science, and business. Depending on the purpose of the product, UX may also involve content design disciplines such as communication design, instructional design, and game design. The subject matter of the content may also warrant collaboration with a subject-matter expert on planning the UX from various backgrounds in business, government, or private groups. More recently, content strategy has come to represent a sub-field of UX.

---

**Graphic designers**

Graphic designers focus on the aesthetic appeal of the design. Information is communicated to the users through text and images. Much importance is given to how the text and images look and attract the users. Graphic designers have to make stylistic choices about things like font color, font type, and image locations. Graphic designers focus on grabbing the user's
attention with the way the design looks. Graphic designers create visual concepts, using computer software or by hand, to communicate ideas that inspire, inform, and captivate consumers. They develop the overall layout and production design for various applications such as advertisements, brochures, magazines, and corporate reports.[25]

**Visual designers**

The visual designer (VisD) ensures that the visual representation of the design effectively communicates the data and hints at the expected behavior of the product. At the same time, the visual designer is responsible for conveying the brand ideals in the product and for creating a positive first impression; this responsibility is shared with the industrial designer if the product involves hardware. In essence, a visual designer must aim for maximum [usability](#) combined with maximum desirability.[26]

**Interaction designers**

Interaction designers (IxD) are responsible for understanding and specifying how the product should behave. This work overlaps with the work of both visual and industrial designers in a couple of important ways. When designing physical products, interaction designers must work with industrial designers early on to specify the requirements for physical inputs and to understand the behavioral impacts of the mechanisms behind them. Interaction designers cross paths with visual designers throughout the project. Visual designers guide the discussions of the brand and emotive aspects of the experience, Interaction designers communicate the priority of information, flow, and functionality in the interface.[27]

**Testing the design**

Usability testing is the most common method used by designers to test their designs. The basic idea behind conducting a usability test is to check whether the design of a product or brand works well with the target users. While carrying out usability testing, two things are being tested for: Whether the design of the product is successful and if it is not successful, how can it be improved. While designers are testing, they are testing the design and not the user. Also, every design is evolving. The designers carry out usability testing at every stage of the design process.[28]

**Benefits**

A great User Experience Design will ensure that the product, website, app or interface on offer is designed to make the lives of the users as simple as possible which can lead to more sales and greater customer retention[29].

User experience design is integrated into software development and other forms of application development to inform feature requirements and interaction plans based upon the users' goals. Every new software introduced must keep pace with the rapid technological advancements. The benefits associated with integration of these design principles include:

- Avoiding unnecessary product features
- Simplifying design documentation and customer-centric technical publications
- Improving the usability of the system and therefore its acceptance by customers
- Expediting design and development through detailed and properly conceived guidelines
- Incorporating business and marketing goals while protecting the user's freedom of choice

**See also**

- Action research
- Activity-centered design
- Chief experience officer (CXO)
- Component-based usability testing
References


Further reading


Why Crunch Modes Doesn't Work: Six Lessons

There's a bottom-line reason most industries gave up crunch mode over 75 years ago:
It's the single most expensive way there is to get the work done.

by Evan Robinson

Executive Summary
When used long-term, Crunch Mode slows development and creates more bugs when compared with 40-hour weeks.

More than a century of studies show that long-term useful worker output is maximized near a five-day, 40-hour workweek. Productivity drops immediately upon starting overtime and continues to drop until, at approximately eight 60-hour weeks, the total work done is the same as what would have been done in eight 40-hour weeks.

In the short term, working over 21 hours continuously is equivalent to being legally drunk. Longer periods of continuous work drastically reduce cognitive function and increase the chance of catastrophic error. In both the short- and long-term, reducing sleep hours as little as one hour nightly can result in a severe decrease in cognitive ability, sometimes without workers perceiving the decrease.

Introduction
In the aftermath of ea_spouse's post on LiveJournal, quality-of-life conversations in the game development business have taken on a new life and a new urgency. Ea_spouse received thousands of comments to her original post — followed quickly by major media coverage. Thousands of people around the net participated in a vast, spontaneous discussion that explored issues like mandatory overtime, productivity, job portability, laziness, unionization, lawsuits and the general evil of corporations.

I've spent 20 years developing and managing software projects. Every year that passed — and every project I worked on — fueled my growing conviction that Crunch Mode is grossly, destructively, expensively inefficient. It's common sense that the more hours people work, the less productive they become. But, over time, I noticed that the productivity losses that result from working too many extra hours start taking a bigger toll faster than most software managers realize. As I dug around, I was stunned to discover that I was hardly the first one to figure this out: my observations have been common knowledge among industrial engineers for almost a century.

I've amassed a personal collection of source information over the past 15 years, this summary mainly includes information that you can readily find on the Web. I don't want you to take my word for it; I want you to be able to go out and read the original source material for yourself.

The History
In 1908 — almost a century ago — industrial efficiency pioneer Ernst Abbe published in *Gessamelte Abhandlungen* his conclusions that a reduction in daily work hours from nine to eight resulted in an increase in total daily output. (Nor was he the first to notice this. William Mather had adopted an eight-hour day at the Salford Iron Works in 1893.)

In 1909, Sidney J. Chapman published *Hours of Labour*, in which he described long-term variation in worker productivity as a function of hours worked per day. His conclusions will be discussed in some detail below.

When Henry Ford famously adopted a 40-hour workweek in 1926, he was bitterly criticized by members of the National Association of Manufacturers. But his experiments, which he'd been conducting for at least 12 years, showed him clearly that cutting the workday from ten hours to eight hours — and the workweek from six days to five days — increased total worker output and reduced production cost. Ford spoke glowingly of the social benefits of a shorter workweek, couched firmly in terms of how increased time for consumption was good for everyone. But the core of his argument was that reduced shift length meant more output.

I have found many studies, conducted by businesses, universities, industry associations and the military, that support the basic notion that, for most people, eight hours a day, five days per week, is the best sustainable long-term balance point between output and exhaustion. Throughout the 30s, 40s, and 50s, these studies were apparently conducted by the hundreds; and by the 1960s, the benefits of the 40-hour week were accepted almost beyond question in corporate America. In 1962, the Chamber of Commerce even published a pamphlet extolling the productivity gains of reduced hours.

But, somehow, Silicon Valley didn't get the memo. Ea_spouse writes:

"The current mandatory hours are 9am to 10pm — seven days a week — with the occasional Saturday evening off for good behavior (at 6:30pm). This averages out to an eighty-five hour work week [sic]."

Actually, working 9am to 10pm, six days a week, plus 9am to 6:30pm one day a week comes out to (6 * 13 = 78 + 9:30 = ) 87.5 hours per week — but after that many hours, who's still counting?
Electronic Arts is no different than many high-tech companies in this regard. For them — and anyone else who wants to increase their employees' productivity and sanity — let's take a look at some of the assumptions management makes regarding hours, output, efficiency, and production costs; and see how a century of industrial research has conclusively, consistently proven those assumptions wrong.

What Management Wants
What is management is trying to achieve when it sends employees off on death marches? Do we honestly believe that the CEO of EA is happy that people are in the office 24/7 working their asses off?

Management wants to achieve maximal output from employees — they want to produce a (good) product as cheaply as possible. They also want to avoid hiring extra resources that increase the cost of the finished goods unless absolutely necessary. On the surface of it, Crunch Mode looks like the most obvious and logical way to reconcile these two desires.

Assuming output can be measured in discrete units, a manager who hasn't read the research may reason that if someone produces, say, 16 units of output in eight hours, they should produce 18 units in nine hours and 20 units in ten hours. To express that view as a simple equation, we can write:

\[ O = X/Y * t \]

Where \( O \) is total output, \( X \) is the given output during a benchmark number of hours, designated by \( Y \), and \( t \) is the actual number of hours worked. In this hypothetical situation, increasing time \( t \) is the simplest way to increase output \( O \).

That assumption may be valid in the limited case where the hours of work are extended over a brief period, for example, to meet a looming deadline. But research — and long experience in other industries — have shown that the limits to such overtime spurts are reached sooner than most people realize. And when those limits are reached, the spurts turn into bogs.

Hourly Productivity Is Important
A more realistic view of worker output would take into account the changes in hourly output that result from a change in the length of the working day. Those changes result mainly from two sources: simple physical and mental fatigue that occurs in the later hours of a long day, and accumulated physical and mental fatigue that builds up over an extended period of long working days.

This more complex view can be represented by the following equation:

\[ O = P(t_1, t_2, t_3 ... t_n) \]

Where \( O \) is total output and \( P() \) represents the changes in hourly productivity that occur over times \( t_1 \cdot t_n \). In this equation \( P() \) is a function, not a constant. \( P() \) will vary by worker, because some workers produce more than others. \( P() \) will also vary by hour, because humans are not machines and do not do exactly the same amount of work in hour 14 of a job as they do in hour 1. Finally, \( P() \) will vary according to the recent history of the worker, because people don’t work as well the morning after a late night as they do the morning after a good night's sleep.

Sidney J. Chapman's Hours of Labour (1909), included (roughly) the following diagram:

[Diagram of worker productivity over time]

Depicting \( P \), the "long-period variations (with the length of the working day) of the marginal value of a fixed quantity of labour" \( OX \) is increasing hours worked in a day and \( OY \) is increasing value. If \( On \) hours are worked, the total value produced is the area \( Onda \). For a lengthier discussion, see http://www.worklessparty.org/timework/chapman.htm. Observe that the height of the curve \( P \) represents worker productivity (output per unit time at a given number of hours worked per day).

Astute readers will note that there is a point, \( b \), where working more hours doesn’t create more value. In fact, after \( b \), each additional hour worked produces negative value. How can this be?

Chapman's diagram of the work curve assumes that a working day of a given length is maintained over a considerable period of time. Thus it incorporates both simple and accumulated fatigue into its model. At first the declines in output per hour simply reflect the effects of fatigue on both quantity and quality of work performed toward the end of a given day. But eventually daily fatigue is compounded by cumulative fatigue. That is, any additional output produced during extended hours today will be more than offset by a decline in hourly productivity tomorrow and subsequent days.

Even during a single "day" of extreme duration, output may come to a standstill as an exhausted employee becomes unable to function. Or output can turn negative as stupefied employees commit catastrophic errors that destroy previously completed work or capital.

In factory terms, a worker’s production rate decreases over time. A worker who is creating 10 widgets/hour at the beginning of a shift may be producing only 6/hr at the end of the shift, having peaked at 12/hr a couple of hours in. Over time, the worker works more slowly, and makes more mistakes. This combination of slowdown and errors eventually reaches a point of zero productivity, where it takes a very long time to produce each widget, and every last one is somehow spoiled. Assembly-line managers figured out long ago that when this level of fatigue is reached, the stage is set for spectacular failure-events leading to large and costly losses – an expensive machine is damaged, inventory is destroyed, or a worker is seriously injured.

In terms of knowledge workers, a programmer produces more good code and fewer bugs when well-rested. We take the first hour or so of the day getting into the groove. The next few hours tend to be our best ones. Later in the day, as we get tired, we get less done per hour — it takes a long time to fix a simple bug or add a simple feature that we would have handled in minutes earlier in the day. Pushed just a little farther — and it seems that much of the computer entertainment industry is working at this extreme most of the time — an overtired IT worker may trash valuable files requiring extra work to restore backups or have an accident on the way home that takes her offline for months.
Lesson One, then, is this: *Productivity varies over the course of the workday, with the greatest productivity occurring in the first four to six hours. After enough hours, productivity approaches zero; eventually it becomes negative.*

Where's The Break-Even?

If productivity essentially decreases over a working day, and working lots of hours results in reduced productivity, how do we establish a method to maximize total output, and figure out where the break-even point lies?

Unfortunately, quantifying knowledge worker output is a hard problem. I would love to be able to give a simple equation you can plug a few numbers into and pull out the magic number of hours each person should work to maximize their output. I can't, because even when such equations finally exist, it will be impossible to find and agree on the basic numbers to plug into them. Common programming measurements, like lines of code and function points are either easy to collect and of questionable value or difficult to define and collect. Useful measures like number of bugs created and number of bugs fixed are viewed with suspicion that they will be used unfairly in annual reviews (or gamed by clever programmers in anticipation of annual reviews or performance bonuses).

Artist output is easier by some measures (number of models or images) and as difficult by some others (subjective quality, look and feel, complexity of model).

Tester output is easy in one sense (number of unique bugs found), expensive in a second (code coverage), and extremely hard in a third (percentage of total bugs found).

Overall, most companies seem to have fallen to a least-common-denominator measure of team output. Either the game ships and sells — or it doesn't. While this is indeed the metric that matters most to shareholders, it's not terribly useful as a measure of productivity, especially daily or hourly productivity.

Lesson Two, then is this: *Productivity is hard to quantify for knowledge workers.*

So we are forced to draw analogies from other industries.

From the Work Less Institute of Technology, *Psychophysics in Cyberia* (written in response to the ea_spouse posting):

"It was over a century ago that Dr. Ernst Abbe conducted his observations on working time and output at the Zeiss Optical Works in Jena, Germany. Dr. Abbe, director of the plant, reduced the daily hours of work from 9 to 8 and kept careful records of daily output per worker before and after the change. What he found confirmed observations from throughout the 19th century: a moderate reduction in working time increased total output. In The Economics of Fatigue and Unrest, Philip Sargent Florence summed up the accumulated evidence to the 1920's:"

"Reduction from a 12-hour to a 10-hour basis results in increased daily output; further reduction to an 8-hour basis results in at least maintaining this increased daily output; but further reductions while increasing the hourly rate of output, seems to decrease the total daily output."

Hugo Münsterberg’s 1913 *Psychology and Industrial Efficiency*:

"...Ernst Abbe, the head of one of the greatest German factories, wrote many years ago that the shortening from nine to eight hours, that is, a cutting-down of more than 10 per cent, did not involve a reduction of the day's product, but an increase, and that this increase did not result from any supplementary efforts by which the intensity of the work would be reinforced in an unhygienic way. This conviction of Abbe still seems to hold true after millions of experiments over the whole globe."

From the Work Less Party, Tom Walker’s *Prosperity Covenant*:

"That output does not rise or fall in direct proportion to the number of hours worked is a lesson that seemingly has to be relearned each generation. In 1848, the English parliament passed the ten-hours law and total output per-worker, per-day increased. In the 1890s employers experimented widely with the eight hour day and repeatedly found that total output per-worker increased. In the first decades of the 20th century, Frederick W. Taylor, the originator of "scientific management" prescribed reduced work times and attained remarkable increases in per-worker output.

In the 1920s, Henry Ford experimented for several years with work schedules and finally, in 1926, introduced a five day, 40 hour week for six days pay. Why did Ford do it? Because his experiments showed that workers in his factories could produce more in five days than they could in six. At every step along the way — in the 1840s, the 1890s and the 1920s — the consensus of business opinion insisted that shorter hours would strangle output and spell economic ruin."

Lesson Three is this: *Five-day weeks of eight-hour days maximize long-term output in every industry that has been studied over the past century. What makes us think that our industry is somehow exempt from this rule?*

What About Short-Term Output?

If 40-hour weeks offer the most reasonable long-term arrangement for maximizing output, can we expect to get short-term gains from short periods of longer workdays or extended workweeks?
In a word, briefly. You can get more work out of more hours for several days to a couple of months, depending upon how much longer the workday is. It is intuitively obvious that a worker who produces one widget per hour during an eight-hour day can produce somewhere between eight and 16 widgets during a 16-hour day. As we've seen, that's the essential logic behind Crunch Mode's otherwise inexplicable popularity. But worker productivity is largely dependent upon recent history. From the Executive Summary of Scheduled Overtime Effect on Construction Projects, published by The Business Roundtable in 1980:

Where a work schedule of 60 or more hours per week is continued longer than about two months, the cumulative effect of decreased productivity will cause a delay in the completion date beyond that which could have been realized with the same crew size on a 40-hour week.

Productivity drops when working 60-hour weeks compared with 40-hour weeks. Initially, the extra 20 hours a week makes up for the lost productivity and total output increases. But the Business Roundtable study states that construction productivity starts to drop very quickly upon the transition to 60-hour weeks. The fall-off can be seen within days, is obvious within a week...and just keeps sliding from there. In about two months, the cumulative productivity loss has declined to the point where the project would actually be farther ahead if you'd just stuck to 40-hour weeks all along.

(The same report cites studies that show total output while working eight-hour days is either 16% or 20% higher than total output working 9-hour days.)

So, yes, Crunch Mode can increase output over the short term. But, at 60 hours per week, in no case should "the short term" be defined as anything more than eight weeks long. At that point, the costs strongly begin to outweigh the advantages. Not only have you lost all the gain those increased hours bought; you've also got tired, angry, burned-out workers. When you return them to a 40-hour week, their output will be sub-par for some time while they recover.

At 87.5 hours per week? Lacking hard data, I would estimate that productivity would drop to under 50% of baseline within a month. An extra 47.5 hours per week (more than double "normal" hours) would provide a large initial surge of extra output.

Lesson Four is this: At 60 hours per week, the loss of productivity caused by working longer hours overwhelms the extra hours worked within a couple of months.

The Sleep Factor

There's another, shorter window that needs consideration in assessing the useful limits of Crunch Mode. That is: How long can someone be productive if they're not getting enough sleep?

Colonel Gregory Belenky, the Director of the Division of Neuropsychiatry at Walter Reed Army Institute of Research, does research for the Pentagon on maximizing the productivity and alertness of soldiers under combat conditions. In his 1997 paper, Sleep, Sleep Deprivation, and Human Performance in Continuous Operations, he found that:

Laboratory studies show that mental work declines by 25% during each successive 24 hours of continuous wakefulness. Sleep-deprived individuals are able to maintain accuracy on cognitive tasks, but speed declines as wakefulness is extended.

... In our study, FDC [artillery Fire Direction Center — ER] teams from the 82nd Airborne division were tested during simulated continuous combat operations lasting 36 hours. Throughout the 36 hours, their ability to accurately derive range, bearing, elevation, and charge was unimpaired. However, after circa 24 hours they ... no longer knew where they were relative to friendly and enemy units. They no longer knew what they were firing at. Early in the simulation, when we called for simulated fire on a hospital, etc., the team would check the situation map, appreciate the nature of the target, and refuse the request. Later on in the simulation ... they would fire without hesitation regardless of the nature of the target.

... At 15 days into the simulation the 4 hour sleep/night battery is firing less than a third of the rounds that the 7 hour sleep/nights battery is firing.

Lesson Five is this: Continuous work reduces cognitive function 25% for every 24 hours. Multiple consecutive overnights have a severe cumulative effect.

Cognitive Decay and Error Rates

One of the biggest productivity sinks created by Crunch Mode is the increase in the number of errors produced. While most errors will be easily fixed, there will be some that could cost all of the output you’ve gained by crunching. The longer you crunch, the greater your odds of creating a big, expensive, schedule-busting monster.

Programmers, artists, and testers aren’t paid for their bulging muscles and phenomenal ability to move mass from point A to point B. They’re paid for their brains. Longer hours or, especially, insufficient sleep (as little as 1-2 hours less per night) does serious damage to their ability to use those brains productively.

Hugo Münsterberg’s 1913 Psychology and Industrial Efficiency:

It has been well known for a long while how intimate the relations are between fatigue and industrial accidents. ... it can be traced everywhere that in the first working hours in which fatigue does not play any considerable role, the number of accidents is small, and that this number sinks again after the long pauses.

Colonel Belenky points out that the consequences to soldiers of losing as little as one hour of sleep per night include "[r]educed ... higher order mental abilities that sustain situational awareness...[r]educed individual and unit effectiveness, errors, accidents..."
It’s a good thing knowledge workers rarely have to worry about "friendly fire."

From Sustained Reduced Sleep Can Have Serious Consequences:

"In a study on the effects of sleep deprivation, investigators at the University of Pennsylvania found that subjects who slept four to six hours a night for fourteen consecutive nights showed significant deficits in cognitive performance equivalent to going without sleep for up to three days in a row. Yet these subjects reported feeling only slightly sleepy and were unaware of how impaired they were. [emphasis mine — ER]"

The Los Angeles Times, Sleepy Medical Interns Called a Road Hazard, by Karen Kaplan, January 13, 2005:

Studies have shown that being awake for 21 hours impairs drivers as much as having a blood-alcohol concentration of 0.08, which is the legal limit for noncommercial drivers in the U.S.

It’s ironic. Most software companies will fire an employee who routinely shows up drunk for work. But they don’t think twice about putting the fate of this year’s silver bullet project into the hands of people who are impaired to the point of legal drunkenness due to lack of sleep. In fact, they will demand that these people work to the point of legal impairment as a condition of continued employment.

The risks are real — and the errors made can be truly catastrophic. From The Promise of Sleep by Dr. William Dement, pp 51-53:

The night of March 24, 1989 was cold and calm, the air crystalline, as the giant Exxon Valdez oil tanker pulled out of Valdez, Alaska, into the tranquil waters of Prince William Sound. In these clearest of possible conditions the ship made a planned turn out of the shipping channel and didn’t turn back in time. The huge tanker ran aground, spilling millions of gallons of crude oil into the sound. ... In its final report, the National Transportation Safety Board (NTSB) found that sleep deprivation and sleep debt were direct causes of the accident. ... The direct cause of America’s worst oil spill was the behavior of the third mate, who had slept only 6 hours in the previous 48 and was severely sleep deprived.

The final report of the Rogers Commission (on the Space Shuttle Challenger accident) said that the decision to launch made during a critical teleconference was flawed. The Human Factors Analysis section suggests that lack of sleep “may have contributed significantly.”

If the bit rot caused by sleep deprivation can lose battles, kill patients, beach oil tankers, and blow up space shuttles, consider what it can do to your $15 million game project.

Lesson Six is this: Error rates climb with hours worked and especially with loss of sleep. Eventually the odds catch up with you, and catastrophe occurs. When schedules are tight and budgets are big, is this a risk you can really afford to take?

What’s It All Mean?

It comes down to productivity. Workers can maintain productivity more or less indefinitely at 40 hours per five-day workweek. When working longer hours, productivity begins to decline. Somewhere between four days and two months, the gains from additional hours of work are negated by the decline in hourly productivity. In extreme cases (within a day or two, as soon as workers stop getting at least 7-8 hours of sleep per night), the degradation can be abrupt.

Many of the studies quoted above come out of industrial environments, and it may be argued that the more creative mental work of programmers, artists, and testers is fundamentally different. In fact, it is different, and Colonel Belenky explicitly addresses that:

In contrast to complex mental performance, simple psychomotor performance, physical strength and endurance are unaffected by sleep deprivation.

The ability to do complex mental tasks degrades faster than physical performance does. Among knowledge workers, the productivity loss due to excessive hours may begin sooner and be greater than it is among soldiers, because our work is more affected by mental fatigue.

When ea_spouse wrote:

The current mandatory hours are 9am to 10pm — seven days a week — with the occasional Saturday evening o for good behavior (at 6:30pm).

she was telling us that the entire team her husband worked on was working at far less than their optimal productivity. They had been working 60+ hour weeks for months already, before management tried to kick them into an 87.5-hour-per week super crunch.

In most times, places, and industries over the past century, managers who worked their employees this way would have been tagged as incompetent — not just because of the threat they pose to good worker relations, but also because of the risk their mismanagement poses to the company’s productivity and assets. A hundred years of industrial research has proven beyond question that exhausted workers create errors that blow schedules, destroy equipment, create cost overruns, erode product quality, and threaten the bottom line. They are a danger to their projects, their managers, their employers, each other, and themselves.

Any way you look at it, Crunch Mode used as a long-term strategy is economically indefensible. Longer hours do not increase output except in the short term. Crunch does not make the product ship sooner — it makes the product ready later. Crunch does not make the product better — it makes the product worse. Crunch raises the odds of a significant error, like shipping software that erases customer’s hard drives, or deleting the source tree, or spilling Coke into a server that hasn’t been backed up recently, or setting the building on fire. (Yes, I’ve seen the first three of these actually happen in the last, bleary days of Crunch Mode. The fourth one is probably only a matter of time.)

Managers decide to crunch because they want to be able to tell their bosses “I did everything I could.” They crunch because they value the butts in the chairs more than the brains creating games. They crunch because they haven’t really thought about the job being done or the people doing it. They
crunch because they have learned only the importance of appearing to do their best to instead of really doing their best. And they crunch because, back when they were programmers or artists or testers or assistant producers or associate producers, that was the way they were taught to get things done.

But it’s not the only way. In fact, the literature shows, over and over again, that it is the very worst way. And that’s the bottom-line reason most industries gave up crunch mode over 75 years ago. Managers, shareholders and employees all stand to benefit from time-tested management practices that will deliver better products, sooner, less expensively -- and with less wear and tear on human resources and public reputations.

Acknowledgments
I’d like to specifically acknowledge:

- Tom Walker of the Work Less Institute for his help in providing documents and reviewing this article;
- Sara Robinson for multiple reviews and revisions, not to mention putting up with incessant talk about dusty old research;
- The IGDA for providing me a bigger soapbox.

Appendix: Collected Sources
As I noted, almost all the documents cited here are available on the Web. While much more data is available in the print literature, I more or less deliberately selected from online studies that the reader can access and use immediately. Here is a collected list of documents I have either quoted from or consulted extensively.

- Ea_spouse
- Psychology and Industrial Efficiency, Hugo Muensterberg, 1913, available at Classics in the History of Psychology, maintained by Christopher D. Green, York University, Toronto, Canada.
- Prosperity Covenant, Tom Walker.
- Samuel Crowther’s interview with Henry Ford, World’s Work, 1926, pp 613-616.
- Sleep, Sleep Deprivation, and Human Performance in Continuous Operations: Colonel Gregory Belenky, Director of the Division of Neuropsychiatry, Walter Reed Army Institute of Research, U.S. Army Medical Research and Materiel Command.
- Sustained Reduced Sleep Can have Serious Consequences, Linda Cook, NINR, March 2003.
- Sleepy Medical Interns Called a Road Hazard, Los Angeles Times, Karen Kaplan, January 13, 2005. Archived at Mischievous Ramblings
- Mischievous Ramblings, Evan Robinson, “It’s Not Just Abusive, It’s Stupid!”
- Mischievous Ramblings, Evan Robinson, “Can People Really Program 80 Hours a Week?”
- Mischievous Ramblings, Evan Robinson, “Staying Awake”

IGDA Resources / Links

- “Quality of Life in the Game Industry: Challenges and Best Practices” white paper
- IGDA open letter: Quality of Life Issues are Holding Back the Game Industry
- Additional Quality of Life resources can be found in the Resources section

Author Bio
Evan Robinson started in the game business at 19 as a developer for TSR. By 22, he was building computer games as an independent developer for EA. In the two decades since then, he has been a grunt programmer, lead engineer, technical director, director of engineering, process consultant, and technical auditor for some of the industry’s best-known companies. Evan’s publications and frequent GDC presentations -- starting in the conference’s early years -- have established him as one of the industry’s voices on best software engineering practices and profitable programmer management. He lives in Vancouver, BC.

Copyright © 2005 Evan Robinson.

The opinions expressed in this article do not necessarily represent the IGDA.